TECHMIG A Layout Tool for Technology Migration

by

Pradip Kumar Kar

EE
1998
M
KAR
TEC

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

MARCH 1998

TECHMIG A Layout Tool for Technology Migration

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Pradip Kumar Kar

to the

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

March 1998

E-1998-M-KAR-TEC

Entered in system

CERTIFICATE

This is to certify that the work contained in the thesis entitled "**TECHMIG A Layout Tool For Technology Migration**" by **Pradip Kumar Kar** has been carried out under my supervision and this work has not been submitted elsewhere for a degree

[illegible]

Dr D Manjunath
Department of Electrical Engineering
Indian Institute of Technology Kanpur

Acknowledgment

I express my greatest gratitude to mv thesis adviser Dr Subir K Roy for his constant support encouragement and valuable suggestions during the thesis work I would also like to thank him for introducing me to the challenging area of VLSI Design the Physical Design Automation

I am also grateful to ARCUS Technology Limited for funding the fellowship program under which this thesis work has been carried out I would like to thank Dr Sandeep Agarwal and Mr Vijay Kundajji of ARCUS Technology Limited for providing some of the technical information and their valuable suggestion towards the development of the tool

I am also thankful to all mv friends who made my stay memorable at I I T Kanpur and mv special thanks to Ajay Bupesh Shivraj Biplab Neela Manish Basudeb Prabhanjan Bhujabal and Rashmi for their constant encouragement and support

Abstract

With numerous foundry services being made available for implementing systems on *VLSI* chips and with the rapid strides that are being made in the fabrication technology of these chips it is imperative from the point of view of economics to be able to migrate any design from one foundry specific technology to another as well as from a present generation fabrication technology to the next generation fabrication technology within the same foundry As the ability to integrate and pack more devices within a die of silicon increases with every new generation of the fabrication technology the complexity of digital systems realizable in a single chip has also grown by leaps and bounds To circumvent implementation bottlenecks a lot of research work has been carried out to fully automate the design of *VLSI* chips under prespecified area and performance criteria The focus of the present thesis is on an important aspect of *VLSI* chip design known as ***Physical Layout Design***

The basic issue addressed here is the following ***Is it possible to port and reuse existing cells in a particular generation of a fabrication technology to a new, but evolving generation by using the mask layout descriptions of the existing cells?"*** As the creation and validation of the mask layout description of cells in any fabrication technology is a time consuming error prone tedious and costly process it is important to be able to make the best possible use of the existing layout resources accumulated from the initial fabrication process This process is known as ***technology migration***

In this thesis study of some existing software tools and algorithms that have been employed for technology migration has been carried out Some new approaches and the corresponding algorithms for carrying out technology migration has been proposed These algorithms have been incorporated in a technology migrator called ***TECHMIG*** ***TECHMIG*** has been developed as a part of this thesis ***TECHMIG*** takes as its input the mask layout description of an existing cell expressed in the industry standard ***Caltech Intermediate Format (CIF),*** along with a set of design rules specific to a new fabrication technology or a new foundry and a set of user specific design constraints It then retargets the initial cell by producing its mask layout description in ***CIF*** which is appropriate for the targetted fabrication technology or the foundry

Contents

List of Figures

Lıst of Tables

Chapter 1

Introduction

State of art *VLSI* fabrication technologies allows fabrication of devices with geometry smaller than 0 25 micron Advances in *VLSI* fabrication will further lower this value in the near future With smaller geometry the ability to fabricate systems on chip is already being demonstrated by many design houses With the possibility of integrating increasingly complex designs in a single die of Silicon chip it is imperative to improve both design productivity as well as design quality by leveraging existing standard cell libraries developed for commercial purposes and research work A large component of any library developmental work involves physical design and layout of cells or circuit elements for a prescribed set of foundry specific design rules The library development work is in general carried out by a very skilled set of layout designer With typical libraries that are being made available by foundries having more than a few hundred cells and megacells any library development for new *VLSI* fabrication processes and technologies with newer design rules is by and large a very time consuming error prone and tedious activity Each new library development generates very few newer cells but retargets existing cells in older technologies for the new design rules Since the design rules are foundry specific a cell designed for a particular foundry is useless for another foundry which obeys an entirely different set of design rules If we have to fabricate the same cell in the second foundry we have to completely redesign the cell

Generally renewal of ***VLSI*** fabrication process is an evolutionary process with incremental changes being introduced in an existing set of design rules As such a set of layout resources accumulated from an old fabrication process must be best utilized to gain full advantages of the newer fabrication process In this thesis we address this issue Specifically we address the following question ***is it possible to use the full masked version of a cell designed in an older technology to create another cell with identical functionality and resource usage, while maintaining the same layout topology that was present in the original cell?*** This process of exploiting the reusability of layout resources and translating it from one set of design rules to the other is called ***"Technology Migration"***

To improve the design quality and design productivity every manual process must be replaced by a well organized set of automatic/semiautomatic tools Most design houses have a history of physical libraries that have been developed for different foundries and these serve as the starting point of the retargetting work These cell layouts are manually edited and migrated to the target design rules set This task is often very time consuming and manpower intensive The motivation for automation arises due to the following factors

- As fabrication technology advances rapidly it often results in a changed fabrication process The new changes result in a large scale change in the design rules
- Circuit performance based on submicron technology is greatly affected by the signal delays Hence there arises a technical issue as to how to optimize shapes and sizes for different kinds of layout elements

- In order to reduce the turn around time for the custom *VLSI* design it is of practical importance to be able to provide in advance a full set of cell libraries

The present thesis is devoted to addressing this problem and to develop a highly automated *CAD* tool for migrating cells from one foundry to the other In this thesis we restrict ourselves to the problem of technology migration for designs based on the standard cell approach In this chapter we first describe in details the process of technology migration and the basic migrator model that we have developed

1 1 Migration Tools, An Overview

There are two broad categories of methodologies capable of efficiently retargetting layout between various processes These are

- To generate layout inside tools that can retarget the results based on different design rules
- To migrate *GDSII* or *CIF* libraries from one process to another

Each of these methodologies have their advantages and shortcomings In both the methodologies once the migration technology file is written for one process the second process migration will be very fast assuming an evolution and not a revolution in the new *VLSI* fabrication process Layout migration is inappropriate when there are drastic differences between the source and the target process e g migrating from a 2 metal layer process to a 3 metal layer process Following are some of the characteristics of the two different approaches of technology migration cited above

Generating a new layout Here every cell should be generated inside the tool The user needs to be an expert in not only in laying out the cells but also in understanding the specific tool philosophy The final results are not perfect in either methodology but in this case the user has to work within the tool to get the desired results The demerit of this approach is that once out of system back annotation for subsequent improvements to the source is not possible Further it can not be used to migrate old libraries or chips that were not previously developed with this methodology The principal advantage with this approach is that cells/blocks topology can be altered inside the tool

Migrating the GDSII or CIF In this approach one can generate the libraries with any kind of layout editor or else one can use old cell layouts These tools use the standard ***CIF*** or ***GDSII*** formats as input and the user may not have to be a layout expert to use the converter If the result of the conversion is not satisfactory a normal layout editor can be used to change the source The principal shortcoming of this approach is that the cell topology cannot be changed to take the advantage of a different process We adopt this approach for our technology migration tool

Migration tools also fall into two other categories viz ***conversion*** and ***compactor*** tools A conversion tools converts a layout from one process to another A tool such as ***Dracula*** from ***Cadence Design Systems*** can do the conversion, but needs additional programming to do compaction

By contrast a compaction tool accepts an existing layout in form of ***GDSII*** or ***CIF*** format then applies the design rule files for a new process The compactor then shrinks the existing layout to a smaller geometry layout converting the layout in the process In operation the compactor shrinks the cell in horizontal direction until a design rule is broken then performs a similar task in the vertical direction Afterwards it begins to apply additional algorithms to further reduce the cell size We follow the compaction approach rather the conversion approach for migrating the layout In this section we describe the basic technology migrator tool *TECHMIG* developed as a part of this thesis work its user interfaces and its internal structure In subsequent chapters we will describe the process of carrying out layout compaction

1 2 TECHMIG

1 2 1 User Interfaces

The various user interfaces of the ***TECHMIG*** tool is shown in Figure 1 The principal component is the central block ***TECHMIG*** The other blocks are the user interfaces These are described as follows

1 2 1 1 The Input Layout

The basic input to the technology migrator is a description of the original cell to be migrated Almost all the compaction algorithms reported in literature take a symbolic description of the layout However in order to capture the full mask features of a layout the symbolic descriptions are inconvenient Prior to fabrication the cell must be available in the mask layer description format rather the symbolic format The symbolic representation is inconvenient and infeasible in representing some extended regions such as ***n well*** and ***p-well*** in ***CMOS*** technology Thus shape description at mask level rather than the symbolic description was the preferred choice as an input to the tool We assume the standard ***Caltech Intermediate Form (CIF)*** representation of a layout The details of ***CIF*** format are given in [1] However we deviate slightly to follow the Compass tool compatible CIF version

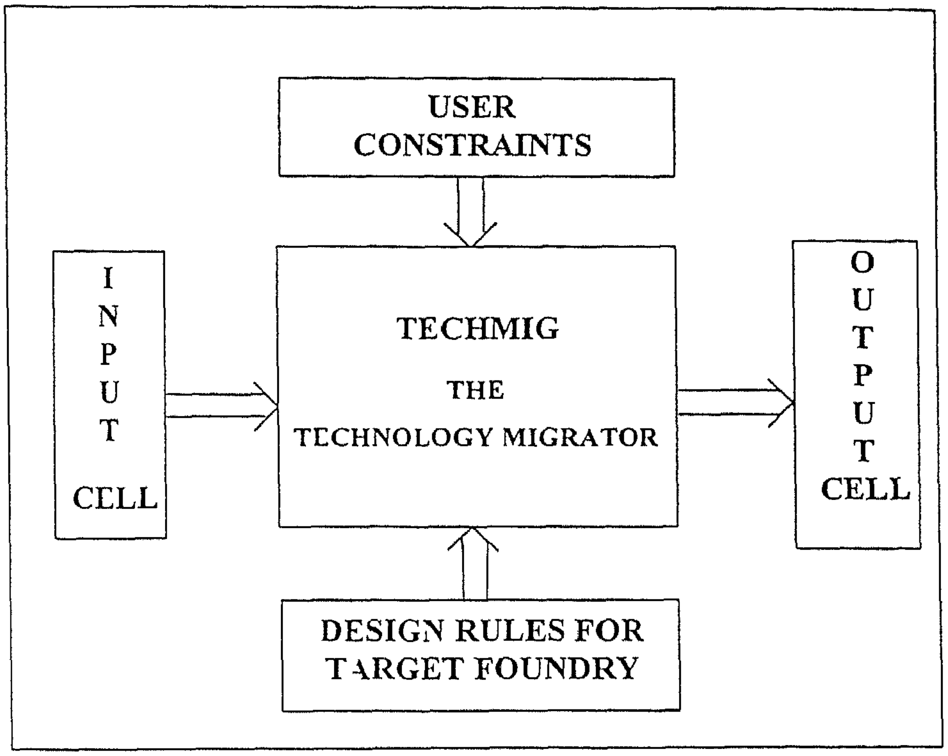


Figure 1 The technology migrator ***TECHMIG*** with its user interfaces

1 2 1 2 The Design Rules

Layout rules or design rules are constraints imposed on the geometry of an integrated circuit layout in order to guarantee that the circuit can be fabricated with an acceptable yield These rules are essential for any design implementation to match the fabrication process specifications Adhering to the design rules is essential for any product to be manufacturable and reliable after fabrication The main objective of a set of design rules is to ensure optimum performance with minimum implementation area without compromising production yield reliability or functionality of the system being integrated into a chip The design rules act as an interface between the circuit designer and the process engineer Circuit designers in general want a tighter smaller design that results in higher performance and higher circuit density The process engineer on the other hand wants a reproducible and high yield process In general design rules represent the best possible compromise between intrinsic device performance and yield In general more conservative the rules are the more likely the circuit will function but with relaxed performance On the other hand if the layout rules are very aggressive it results in better circuit performance due to reduced parasitic and improved characteristics The design rules primarily address two issues

- The geometrical reproduction of features that can be produced by mask making and the lithographic processes
- The interference constraints between different layers

The fundamental unity in the definition of a set of design rules is the ***minimum line width*** which is defined as the minimum mask dimension that can be safely transferred to the semiconductor material It is set by the resolution of the patterning process At least two basic approaches are taken in describing the dimensions in design rules

Scaleable Dimensions The scaleable design rules also called the ***lambda (λ)*** based rule was made popular by Mead and Conway [1] It defines the entire set of design rules as a function of a single parameter called ***lambda (λ)*** which is specific to the particular fabrication process In Mead and Conway [1] the basic unit of length measurement is equal to the fundamental resolution of the process itself ***Lambda*** is defined as the distance by which a geometrical feature on any one layer may stray from another geometrical feature on the same layer or on another layer considering all the processing factors and adding an appropriate safety factor The value of ***lambda*** is set by phenomena associated with the fabrication process such as over etching misalignment between mask levels distortion of the silicon wafer (run out) due to high temperature processing and overexposure and underexposure of resist.

In this method the rules are chosen so that a design is easily ported over a cross section of industrial process Scaling of the minimum dimension is accomplished by simply changing the value of *lambda* For a given process λ is set to a specific absolute value and all design functions are consequently translated into absolute numbers The minimum line width of the process is set to 2λ. For instance for 1 2 micron process λ equals to 0 6 micron The detail description of ***lambda*** based rules is given in [1] and [2]

Absolute Dimensions (microns) Here all the design rules are described as its absolute value For example the minimum width of active diffusion area may be specified as 1 micron Almost all the industries follow this style than following the lambda based rule The reasons behind the adaptability of micron based rule are

- Adopting the lambda based rule exploits the advantages of linear scaling of the circuit elements However as the design complexity increases with simultaneous shrinkage of device size linear scaling can lead to design rule violations Linear scaling is only possible over a limited range of dimensions (for instance between 3 micron and 1 micron) When scaling over larger ranges (for instance into sub micron range) the relations between different layers tend to vary in a nonlinear way that cannot be adequately covered by linear scaling rules

- Scaleable design rules are conservative As they represent a cross section over different technologies they have to represent the worst case rules for the whole set which results in an over dimensioned and less dense design

For these reasons scaleable design rules are avoided by industries As circuit density is the primary goal in industrial design most semiconductor industries tend to use micron rules However the scaling and porting of designs between technologies under these rules is more demanding and has to be done either manually or by using rule The reasons behind the adaptability of micron based rule are

TYPES OF DESIGN RULES

At a sufficiently high conceptual level all *CMOS* processes use the following features

- Two different substrates and or wells *(CNW CPW)*
- Active regions for both p and n transistor as well as substrate taps *(CND CPD)*
- One or more polysilicon layers which serves the purpose of gate of the transistors as well as the interconnection layer *(CP)*
- One or more levels of metal interconnection layer *(CM CM2)*
- Contacts and vias to provide interlayer connections *(CC CC2)*

Detailed description of these layers are given in [3] The design rules reflects the interaction of these layers in the layout plane Broadly we can classify the whole set of design rules into following categories

- Size Rules
- Separation Rules
- Overlap / Extension Rules

Size Rules These rules refer to the minimum width of a wire on a particular layer This is imposed to prevent thin features such as wires being narrow enough to break the connection and also to prevent electromigration The minimum feature size of a device or an interconnect is determined by the line patterning capability of lithographic equipment used for IC fabrication Interconnect lines usually run over a rough surface unlike the smooth surface over which active devices are patterned Consequently the minimum feature size used for interconnects is somewhat larger than that used for active devices The various design rules under this category are given in [3] Figure 2 gives an illustration In Figure 2(a) the minimum required width of a *diffusion* layer is shown and in Figures 2(b) and (c) minimum width for *poly* and *metal1* layer are shown

Separation Rules Different features on the same layers or on interacting layers need to be separated from each other These separation type design rules prevent the separate and isolated features from accidentally short circuiting with each other Figure 2 illustrates some of the spacing rules These spacing rules can also be specified conditionally For example the spacing between two metal layers depends upon the thickness of the wires and the extent to which the two wires run in parallel The required spacing between two wide metal wires is somewhat larger than the required spacing between two narrow metal wires Similarly the required spacing between two parallel and long wires is somewhat higher than the required spacing between two short wires These types of rules which depend upon certain characteristics of the primitives are called the ***conditional rules*** ***TECHMIG*** has the capability of supporting such conditional rules A detailed listing of various spacing rules both conditional and unconditional is given in [3] The separation rules for three different layers are shown in Figure 2

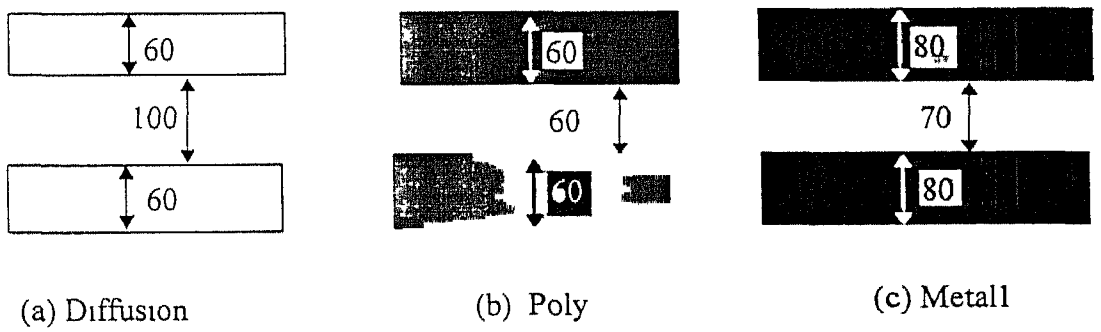


Figure 2 Illustrating size and separation rules for diffusion poly and metal1 layer All dimensions are in hundreds of microns

Overlap/Extension Rules The overlap rules ensure the integrity of the topological features in a layout The extension rules particularly refers to the extension of poly region beyond the diffusion region defining the transistor This prevents the formation of incorrect transistor or short circuited channel In Figure 3(a) a contact overlap rule is shown and in Figure 3(b) the extension rule for a transistor is shown Detailed listings of the various overlap/extension rules are given in [3]

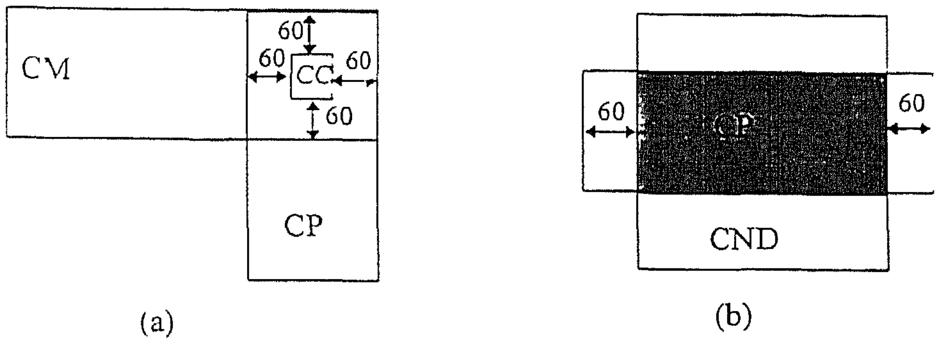


Figure 3 Illustrating overlap and extension rules (a) *CC* *CM* and *CC - CP* overlap rule and (b) *CP* *CND* extension rule Shaded area is the transistor All dimensions are in hundreds of micron

REPRESENTATION OF THE DESIGN RULES

Specific to a fabrication process the number of design rules typically varies from one to a few hundred These are normally specified in plain English by a one sentence description Additionally graphic illustration may be provided for the various cases of applying these rules The common representation of fabrication rules is by a matrix of simple spacing rules between pairs of masks with size and overlap rules stored in separate tables Presently various design houses follow the ***EDIF*** format for design rule description However we follow a uniform framework of design rule description as described in [4] Here we briefly describe the format for design rule description The followings are some of the characteristics of this method of representation

- All rules are formulated in the same manner including space rules size rules overlap rules and extension rules This is essential for the compaction of shape objects
- *Conditional rules* are allowed We support the conditional rules related to some specific topology Provisions are not made to represent other conditional rules related to parasitic resistance capacitance and reliability
- A *linked list* structure instead of a full matrix form is used for the layout rules to reduce the space complexity In matrix form we also have to reserve spaces for noninteracting layers for which no design rule exist

The boundary of a mask shape is a set of lines with each line having two sides namely ***inside*** or ***outside*** The ***inside*** refers the side of a line towards the mask and the ***outside*** is the side not included in the mask Since the rules depend upon both the mask level and the side of the boundary an edge is defined as a boundary line with a side attribute That means

edge = T(mask side)

where ***T*** is the transformation applied to the mask depending upon the side attribute

The required transformation ***T*** is defined as

T(mask Inside) = mask

T(mask Outside) = -mask,

For example on an n diffusion layer ***(CND), CND*** represents an inside edge while ***CND*** represents an outside edge

A simple rule between a pair of masks can be expressed as ***edge1 edge2 Value*** where ***Value*** is the absolute value of the design rule between the corresponding edges in hundreds of micron (standard unit of the dimensions in CIF representation of the layout) Some examples of simple rules are shown in Figure 4 In Figure 4(a) the overlap and size rules are specified and in Figure 4(b) the spacing rules are specified Here ***mask1*** is a contact layer and ***mask2*** is a metal1 layer

The size rule for the metal wire is given as ***CM CM Wmin Wmin*** being the minimum required width of the metal wire If the minimum overlap of a metal layer on the contact is ***Min_Overlap*** the corresponding rule is specified by ***CM CC Min_Overlap*** Similarly the minimum spacing rule between two poly layers is specified as ***CP CP Min_space***

Conditional rules are represented by augmenting the edge description with another attribute For example the rule between two wide metal wires on the metal1 layer can be represented by ***-CM_W CM_W value_w*** and the rule between two narrow metals similarly can be represented by ***CM_N CM_N value_n***

The detailed listing of various types of design rules under these categories (size space overlap) is specified in [3]

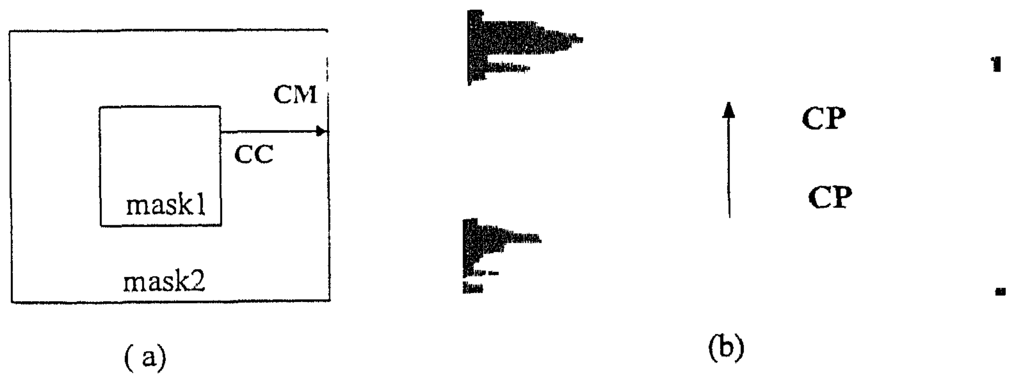


Figure 4 Illustrating the representation of design rules

1 2 1 3 User Constraints

These refer to some special types of rules as per the requirement of the user commonly known as the ***user constraints (UC)*** These include the power bus *(V_{DD}* and *V_{SS})* width connector positions transistor sizes cell height etc pertinent to the cell The fourth type of rules i e the user constraints does not depend on the particular fabrication process rather are specified by the user depending upon his/her requirements These being very specific to a layout are represented in a separate format

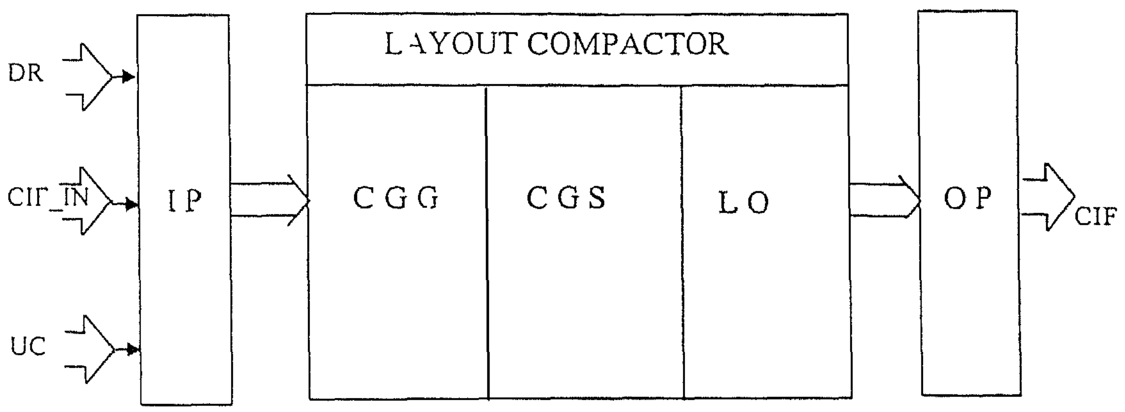
The rules regarding transistor resizing are stored in a table called ***Transistor Resizing Table (TRT)*** which has information regarding the position of the transistor in the original layout it s *W/L* value in old technology and it s *W/L* value in the new technology The other user constraints regarding power bus *(VDD* and *VSS)* width cell height and connector placement are provided by the user at run time

1 2 1 4 Output Layout

As the tool does not change the topology of the original layout the output layout has the same format as the input layout Thus our tool produces the output cell in *CIF* format compatible with Compass *CIF* version

1 2 2 Internal Structure of TECHMIG

The different blocks that the tool consists of are shown in Figure 5 These are described as follows



1 DR Design Rules	2 UC User Constraints
3 CIF_IN Input CIF description of cell	4 I P Input Processor
5 O P Output Processor	6 C G G Constraint Graph Generator
7 C G S Constraint Graph Solver	8 L O Layout Optimizer

Figure 5 Illustrating the internal structure of the technology migrator ***TECHMIG***

1 2 2 1 Input Processor

The input to this block is a ***Compass tool*** compatible *CIF* description of the source layout the design rules files and various user constraints The input processor (I P) reads these information processes it and stores the relevant information in some internal databases like an array of polygons or a linked list etc The representation of design rules has already been described and these are stored in a linked list format Here we give a brief description of the internal representation of the input cell The *I P* handles this task of converting the *CIF* description of the layout to this internal representation

Layout patterns for functional cells are composed of layout elements Each element is represented as a polygon with attributes such as shape size location and the mask layer (such as diffusion metal etc) Such an individual element is also referred to as a ***primitive*** We use the term polygon instead to refer to either a primitive or an element

In almost all published articles the basic layout primitive is restricted to only rectangles Any mask polygon of an arbitrary shape in the Manhattan geometry is sliced into a number of rectangles In our approach we overcome this limitation and instead represent the polygon as a single entity This form of representation helps in manipulating structures with non Manhattan geometry (e g 45 degree bent wires) and results in a constraint graph with minimum number of nodes This accelerates the subsequent steps of involving constraint graph solution and optimization The second advantage is due to the fact that every edge of the polygon contributes to a node in constraint graph so the number of nodes in constraint graphs due to a particular polygon is equal to the number of edges in the polygon However if we take the rectangles as the basic layout entity and slice the polygon into a number of rectangles the number of nodes contributed to the constraint graph depends upon the slicing pattern as a particular polygon can yield many different combination of rectangles for different slicing patterns Optimal slicing of polygon is not well studied in literature and slicing to rectangles is not possible for layouts with non-Manhattan geometry The entire layout is represented as a linked list of polygons The constraint generation step utilizes this linked list representation of the layout to analyze all the aspects of the layout like separation overlap identification of transistors etc

1 2 2 2 Layout Compactor

A *compactor* takes as its input a ***VLSI*** layout and produces as its output an equivalent layout of smaller area The compactor moves components and wires in the layout plane to optimize for two basic goals that the layout should be of as small area as possible and that it should be design rule correct The compactor moves subcells only in the plane preserving the cell topology Auxiliary functionalities such as wirelength minimization automatic jog insertion contact optimization etc are generally incorporated to improve its capability to get a better layout according to specified performance criteria The compaction problem is stated as a minimization problem that is find the minimum of the area function subjected to linear and nonlinear constraints

The compaction algorithms can be classified as ***one dimensional*** and ***two dimensional*** compaction according to the movements of components during the compaction process In one dimensional compaction the individual components are moved only in a single direction either X or Y in a single pass For a complete compaction of a flattened layout two passes of 1D compaction are necessary In 2D compaction a single pass can move a component in both directions

Compactors can also be classified according to the algorithms used to position the components The major techniques of the compaction are the ***constraint graph compaction (CG) virtual grid compaction (VG)*** and ***zone refining (ZR)*** technique The descriptions of these approaches are given in [5], [6] and [7] We use the constraint graph approach as it is the most popular and widely used method. We describe in details this approach in Chapters 2 and 3 The zone refining technique for IC layout compaction is given in [8]

Several one dimensional and two dimensional compaction algorithms are available in literature of which a versatile one is presented in [9] which is based on the solution of longest path In [10] the compaction problem is translated into a ***mixed integer linear programming*** of a special kind and a graph based branch and bound algorithm is used to solve the problem The literature presented in [11] uses an algorithm almost parallel to that presented in [9] except for the extra provision for considering the grid constraints if any The problem of 2D compaction has been shown to be ***NP complete*** in [12] In [12] a ***branch and bound optimization (BBO)*** method is proposed for 2D compaction the objective function being the minimum area of the bounding rectangle The problem of automatic jog insertion during the compaction process is not well studied in literature

We restrict ourselves to alternating passes of 1D compaction and formulate the compaction problem as a linear programming problem We then solve this problem to minimize the bounding rectangle area The total wirelength serves as the auxiliary optimization function In our approach the entire compaction problem is split into the constraint generation constraint solution and optimization phase In block diagram notations these steps are shown in Figure 5 In the first step the various design rules are imposed on the initial layout This requires the construction of the constraint graphs for both the X and the Y direction The constraint generation is the most important step of the compaction process as it requires a detailed analysis of the layout features using computational geometry After imposing the entire set of design rules the optimization problem is modeled as a linear programming problem Using this formulation we search for a solution which leads to a layout with minimum area and which satisfies all the design rules Wire length minimization is performed as a post optimization phase to additionally improve the layout to meet the desired performance specifications Wire length minimization reduces the net parasitic resistance of the layout by decreasing the wire length in highly resistive layers and increasing it in comparatively low resistive layers

Various algorithms for generating the constraints are described in Chapter 2 In Chapter 3 we discuss the various aspects of constraint solving and wire length optimization and the corresponding algorithms

1 2 2 3 Output Processor

In the compactor we translate the layout features and design rules into a graph called constraint graph and then solve the compaction problem using graph theoretic algorithms The final stage in technology migration consists of output processing In this phase the graphs representing the positions of the layout primitives after the optimisation phase are translated into the *CIF* format Thus in this stage we obtain the output cell suitable for a target foundry

Chapter 2

Constraint Generation

2 1 Introduction

One of the basic task in constraint graph based compaction is to translate the design rules into a set of constraints and imposing those constraints to the layout elements These constraints reflect the sizes of the primitives their relative positions in the layout and some specific design rule requirements pertinent to the target fabrication process Keeping all these factors in view we can broadly classify the whole set of constraints into following categories

- Constraints specifying minimum required width of a particular layer
- Constraints to represent minimum required spacing between two interactive layers
- Constraints to represent overlaps and extensions between primitives
- Transistor resizing constraints
- User constraints to fix the cell height power bus width, and connector positions in the layout

In this chapter we describe various algorithms to generate these constraints When the entire set of constraints is represented by a graph the resultant graph is known as a constraint graph Two graphs one to represent constraints in horizontal(X) direction called the horizontal constraint graph *(G_H or HCG)* and the other representing set of constraints in vertical (Y) direction called the vertical constraint graph *(G_V or VCG)* needs to be generated We describe these constraint graphs briefly as follows

The layout is flattened and represented in a two dimensional (XY) plane Each edge of a primitive e g a rectangle is represented as a node in constraint graph A vertical edge contributes to a node in *HCG* and a horizontal edge contributes to a node in *VCG* Any inclined edge is represented as a node in both G_H and in G_V Two nodes in a constraint graph are linked by an arc if there exists any design rule which relates the corresponding edges of the primitives in the layout Thus the two graphs G_H and G_V encode the set of constraints G_H encodes horizontal constraints while G_V encodes vertical constraints Each vertical or inclined edge in the layout is a node x_i in the graph G_H and each horizontal or inclined edge is a node y_i in the graph G_V The graphs are directed acyclic graphs Each directed arc in the graph corresponds to an inequality Each arc (x_i, x_j, W_{ij}) has a weight W_{ij} associated with it representing the constraint between the vertical edges corresponding to the nodes x_i and x_j respectively Implicitly the arc represents the inequality $X_j - X_i \geq W_{ij}$

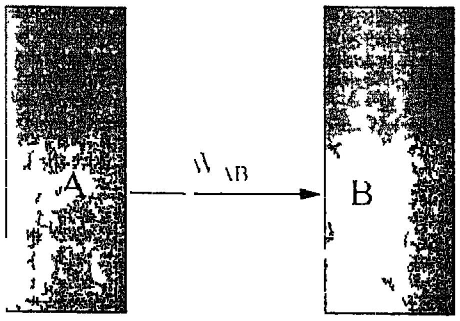
Similarly each arc (x_i, x_j) is associated with a cost parameter C_{ij} that roughly indicates the resistance of associated layer This cost parameter serves as a secondary objective for the compaction problem Exact formulation of cost function and cost parameters are explained in Chapter 3

Formally we define $G_H = \{X\ E_x, W_x, C_x\}$ as a directed weighted graph $X=\{x_i\}$ is the set of nodes in the graph one for each edge of layout and two special nodes S_x *(source)* and T_x *(target or sink)* S_x corresponds to the leftmost edge in any cell in the layout and T_x corresponds to the rightmost edge

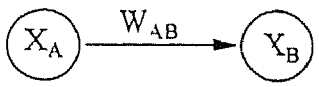
$E = \{ <x_i, x_j> \mid x_i, x_j \in X\}$ is the set of arcs in the graph, one for each inequality

$W_x = \{ W_{ijx} \mid <x_i, x_j> \in E_x, W_{ijx} \in R\}$ is the set of weights specifying the separation between the nodes x_i and x_j

$C_x = \{ C_{ijx} \mid <x_i, x_j> \in E_x, C_j \in R\}$ is the set of cost parameters related to layer resistance

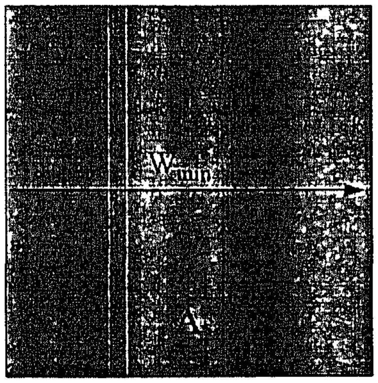


(a)

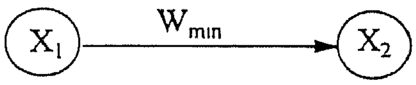


(b)

Figure 6 Illustrating a spacing constraint W_{AB} is the minimum required spacing between primitives A and B



(a)



(b)

Figure 7 Illustrating the minimum width constraints Here W_{min} is the minimum required width of the primitive A

Some examples of constraint graph are shown in Figure 6 and Figure 7 In Figure 6 two primitives A and B are required to be spaced by a distance of at least W_{AB} units Figure 6(b) shows the constraint graph Here the cost value is set zero In Figure 7 for a primitive A its minimum width should be W_{min} units and the layer cost is C units This is represented in graph domain as shown in Figure 7(b)

2 2 Minimum Width Type Constraint Generation

In this section we describe the algorithms to generate constraints that relate to the minimum required width of an entity This is dependent on the type of the layer associated with the entity The minimum required widths for each layer type corresponding to certain fabrication process are given in [3] In software implementation this minimum width type of design rules are stored in a table whose individual entry has three fields

- *Layer* Specifies the characteristic layer for the mask polygon
- W_{min} Specifies the minimum required width for that layer
- *Cost* Specifies the sheet resistance of the layer

In this section we will describe the algorithms to impose the minimum width type constraints to the layout and to add those constraints to the constraint graphs G_H and G_V Following procedures find the minimum width and the sheet resistance of a particular layer

```
Procedure Min_Width (Layer)
Begin
    Curr_entry = first entry of the min_ width  design rule table,
    While (All entries not searched) do
    Begin
```

```
            If (Curr_entry → layer = Layer) return (Curr_entry → $W_{min}$);
            Curr_entry = next entry of the min_width design rule table
      End while
End


Procedure Min_Width (Layer)
Begin
      Curr_entry = first entry of the min_width design rule table
      While ( All entries not searched) do
       Begin
            If (Curr_entry → layer = Layer) return (Curr_entry → Cost)
            Curr_entry = next entry of the min_width design rule table
       End while
End
```

2 2 1 The Scanline Algorithm

To add the minimum width type constraints we take one primitive at a time Depending upon its layer we determine the required minimum width as per the procedure *Min_Width()* Then we detect the edge pairs of the primitive which are to be related by a width constraint Detecting all edge pairs involves running a scan line through the primitive in a direction perpendicular to the direction of constraints e g horizontal scanline running in vertical (Y) direction for adding horizontal width constraints and vertical scanline running in horizontal direction to add vertical width constraints The following is an illustration of the scanline technique to detect all edge pairs which are to be constrained horizontally

A scanline $X'X$ is run in $+Y$ direction in the layout plane An edge of the polygon is inserted to the scanline when its bottom end point is encountered and deleted from the scanline when its top end point is encountered After inserting all edges that corresponds to a particular position (Y coordinate) of the scanline adjacent edge pairs are checked to find out whether the edge pairs are exactly to be constrained If any arbitrary point intermediate to two adjacent points on the scanline remains within the polygon then the corresponding two edges represent the opposite edge pairs of some portions of the polygon and hence have to be constrained

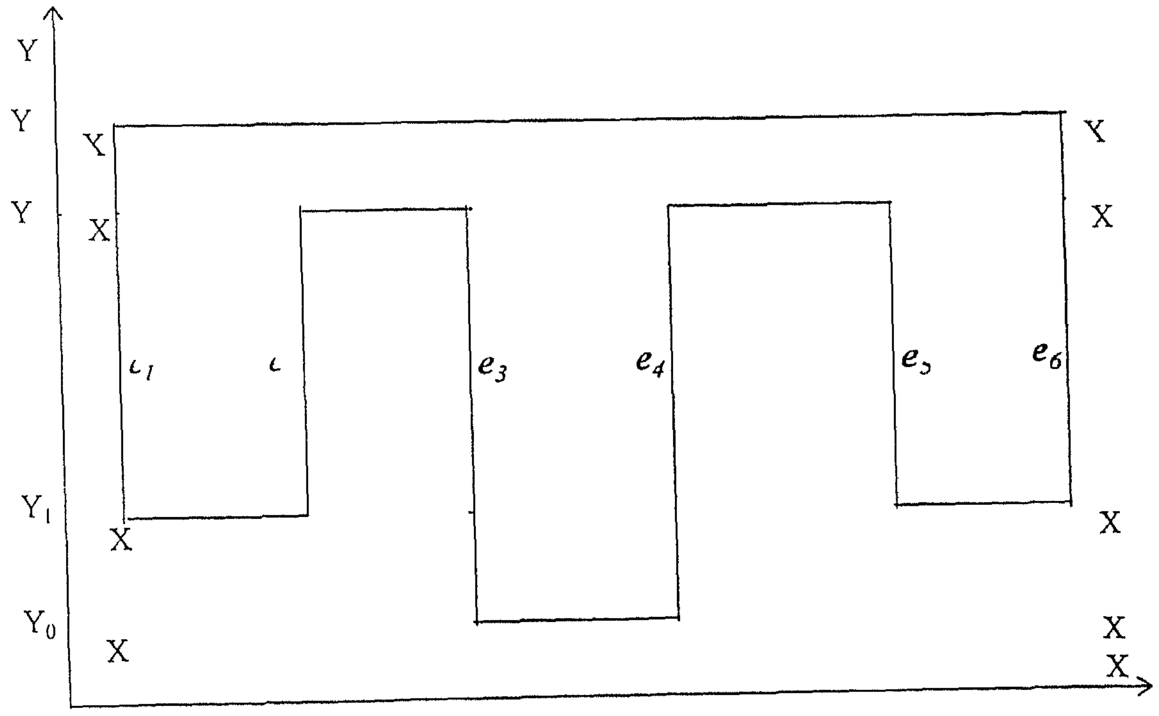


Figure 8 Illustrating the scanline technique for generating minimum width type constraints

In Figure 8 a primitive A is shown which is to be constrained through the size rules in horizontal direction A horizontal scanline $X'X$ is run from the bottom to the top of the polygon Its positions at distinct instances are shown At position Y_0, the edges e_3 and e_4 are inserted to the scan line It is found that these two edges enclose certain portions of the polygon in horizontal direction which means that these two edges are actually to be constrained by the width constraints and hence a width constraint is joined between edges e_3 and e_4 Then the scanline is moved to the position Y_1 where it encounters the bottom end points of the edges e_1, e e_5 and e_6 The edges e_1 e , e_5 and e_6 are inserted to the scanline Adjacent edge pairs e_1 e_2 and e_5 and e_6 are constrained as they enclose some portions of the polygon However the adjacent edge pairs e and e_5 are actually not to be constrained We take this decision by predicting a point intermediate to e and e_3 and checking whether that point remains within the polygon If it lies within the polygon then the edge pairs are actually to be constrained by width constraint At position Y the bottom end points of edges e e_3 e_4 and e_5 are encountered by the scanline hence e_2 and e_3 are deleted from the scanline Similarly at position Y_3 the edges e_1 e_6 are deleted from the scanline Figure 9 shows the constraint graph generated by this algorithm A formal description of our algorithm to generate size constraints in both horizontal and vertical direction corresponding to a layout entity is given below

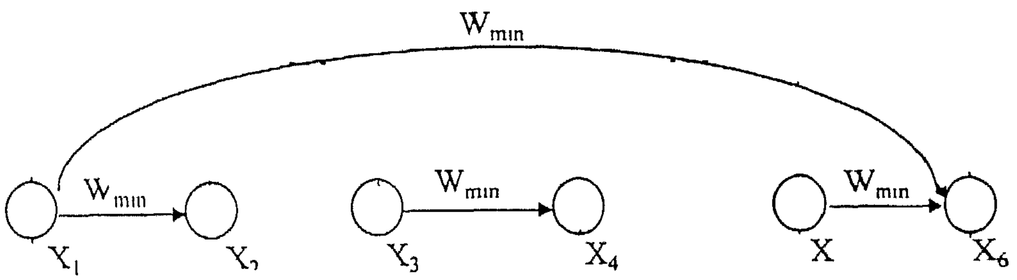


Figure 9 The constraint graph showing ***Min_Width Constraints*** only as genertated by ***scanline*** technique

```
Procedure Add_Min_Width_Constraints_to (entity)
Begin
    $W_{min}$ = Minimum width of layer corresponding to entity,
    if (entity $\rightarrow$ type = box)
    Begin
    Find opposite edge pairs of the box  Join the constraints of weight $W_{min}$
    between the corresponding nodes in the constraint graph
     return
    End if
    if (entity $\rightarrow$ type = polygon)
    Begin
    Step 1  Create an array H[] whose elements are the end points of the
    horizontal and inclined lines  Similarly create an array V[] whose elements
    are the end points of vertical and inclined lines

    Step2  Sort the elements of H[] array according to the following strategies
        •  Take X coordinates of the points as primary key  A point having lower
           X coordinate value has higher priority of being scanned early
        •  Amongst the points with equal X coordinates, those representing right
           end points have higher priority of being scanned  early  (This ensures
           that at a particular position of the scanline the deletion of edges has
           higher priority than the insertion of edges)
        •  Sort the group of points that have the same X coordinates and  all
           points  representing  the left end points in increasing order of Y
           coordinates

  Step 3  Sort the elements of V[] array according to the following strategies
        •  Sort the points in increasing order of Y coordinates
```

- *Amongst the points with equal Y coordinates the top end points have higher priority than the bottom end points*
- *Sort the group of points with equal Y coordinates, and all representing the bottom end points in increasing order of X coordinates*

Step 4 Maintain an array V_sc[ ] that represents a vertical scanline The elements of V_sc represent the points that are currently on the scanline

Step 5 Run the scan line V_sc[ ] from left to right. From H[ ] array select a group of points that have equal X coordinates

```
Label 5 1   for (each point ∈ group)
        Begin
        if (point = left end point)
        Begin
            Insert that point in the scanline V_sc[ ] at proper position such
            that the elements of V_sc[ ] are always in the y-sorted order
        End if
        else replace the corresponding left end point by the current right
        end point
End for
```

step 6 Between every pair of end points P1,P2 $\in$ V_sc[ ] such that (P1 P2,W_{min}) represents a valid width constraint, join a constraint of weight equal to the minimum width of the corresponding layer Delete all right end points from the scanline V_sc[ ]

if (all groups are not selected) select next group and goto Label 5 1

Step7 Maintain an array H_sc[ ] that represents a horizontal scanline The elements of H_sc represent the points that are currently on the scanline

```
Step 8  Run the scan line H_sc[ ] from bottom to the top  From H[ ] array
select a group of points that have equal Y coordinates
Label 8 1   for (each point ∈ group)
Begin
        if (point = bottom end point)
        Begin
         Insert that point in the scanline H_sc[ ] at proper position, such that
         the elements of H_sc[ ] are always in the x_sorted order
         End if
      else replace the corresponding bottom end point by the current  top end
      point
End for

Step 9  Between every pair of end points P1,P2 ∈ H_sc[ ]   such that
(P1 P2 $W_{min}$) represents a valid width constraint join a constraint of weight
equal to the minimum width of the corresponding layer  Delete all top end
points from the scanline H_sc[ ]
if (all groups are not selected)  select next group and goto Label 8 1

End
```

2 3 Spacing Constraint Generation

The generation of spacing constraints is a very time consuming process in mask layout compaction based on constraint graph approach In worst case there are ***O** (N)* constraints where *N* is the no of primitives in the layout One way to limit the number of constraints is not to generate redundant constraints Shadow propagation is one traditional approach to generate the spacing constraints where ***X visibility*** and ***Y visibility*** are checked to determine whether two edges of layout entities are to be constrained by the spacing constraints Two primitives are said to be ***X visible*** if a horizontal line extended from one primitive in *+X* direction meets the other without passing through any extra primitive Synonymously ***Y visibility*** can be defined Concept of ***X visibility*** is shown in Figure 10 Here the primitive ***B*** is ***X visible*** from the primitive *A* while primitive ***C*** and ***D*** are not

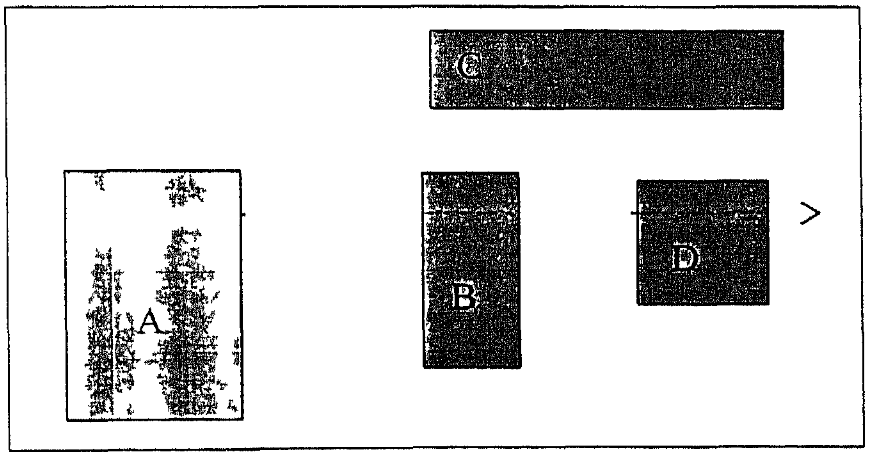


Figure 10 Illustrating the concept of visibility Here ***B*** is ***X visible*** from ***A*** while ***C*** and ***D*** are not

The constraint generation method used should ideally generate an irredundant set of constraints since the cost of solving the constraint graph is proportional to the number of edges of the constraint graph In practice we must trade increased constraint generation time to generate the smaller constraint set for smaller solving time

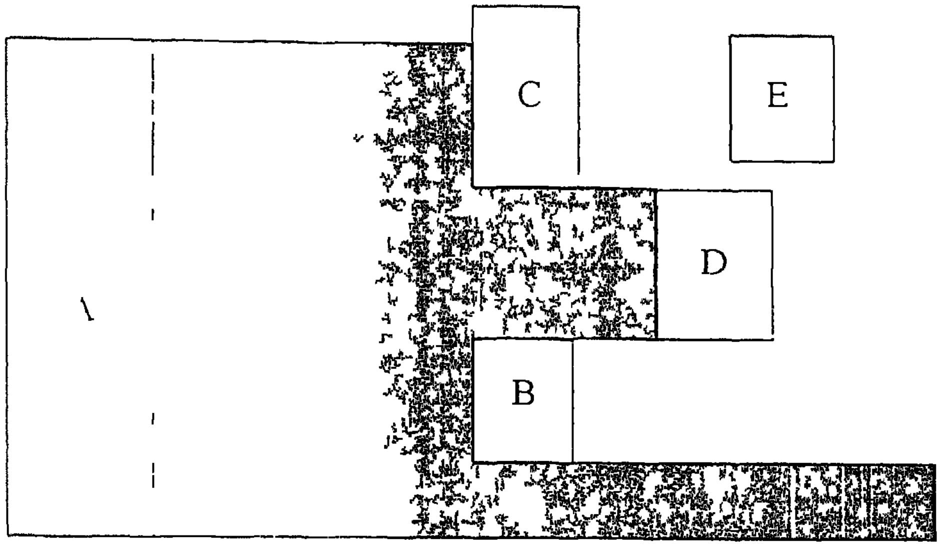


Figure 11 Illustrating *Shadowing* using *X Visibility*

The shadow propagation algorithm examines the positions of the cells to determine what constraints are redundant Referring to Figure 11 we can imagine a region that contains the rectangles that must be constrained against primitive *A* in horizontal direction as falling under a shadow cast from *A* If the shadow falls on a rectangle that rectangle must be constrained against *A* Hence we must generate constraints from *A* to *B* *C* and *D* We need not generate a constraint from *A* to *E* as *E* does not fall in the shadow of *A* However *E* is indirectly being constrained against A through the entity *C* as E falls under shadow of *C* which is in the shadow of *A*

The above mentioned method is good enough for generating a irredundant set of constraints but the only problem with this approach is that it produces an under estimated set of constraints The diagonally interactive entities cannot be constrained by this approach hence this cannot prevent two initially non overlapping primitives from overlapping each other in new technologies Figure 12 is an illustration of this fact

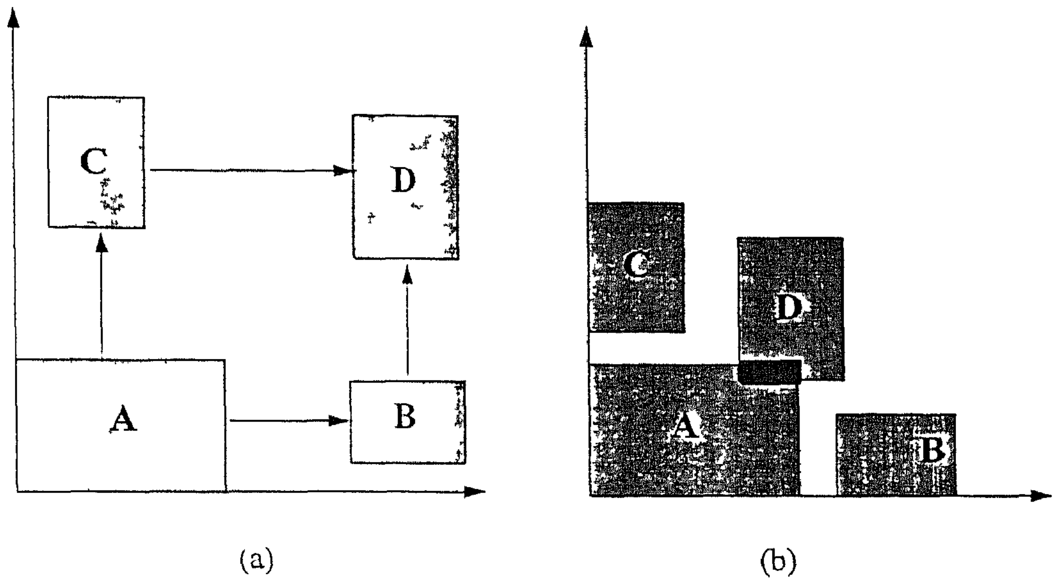


Figure 12 Illustrating the shortcomings of shadowing using X-visibility (a) initial configuration before compaction (b) final configuration after compaction

In Figure 12(a) the initial layout with all its rectangles spaced from each other are shown The arrow represents the vertical and horizontal spacing constraints between the various rectangles as generated by conventional shadow propagation algorithm The rectangles *A* and *D* are initially non overlapping However if the layout is compacted as there is not any constraint between *A* and *D* these two primitives cannot be prevented from being overlapped as shown in Figure 12(b) To ensure the spacing relations in diagonal direction we modify the definition of ***shadow*** and redefine it by casting a shadow at an angle of 45^0 as shown in Figure 13(a) Shadowing in diagonal direction ensures that the primitives in diagonal direction, e g *A* and *D* are constrained and when the layout is compacted the topology can be preserved The basic idea of not generating redundant constraints is as follows

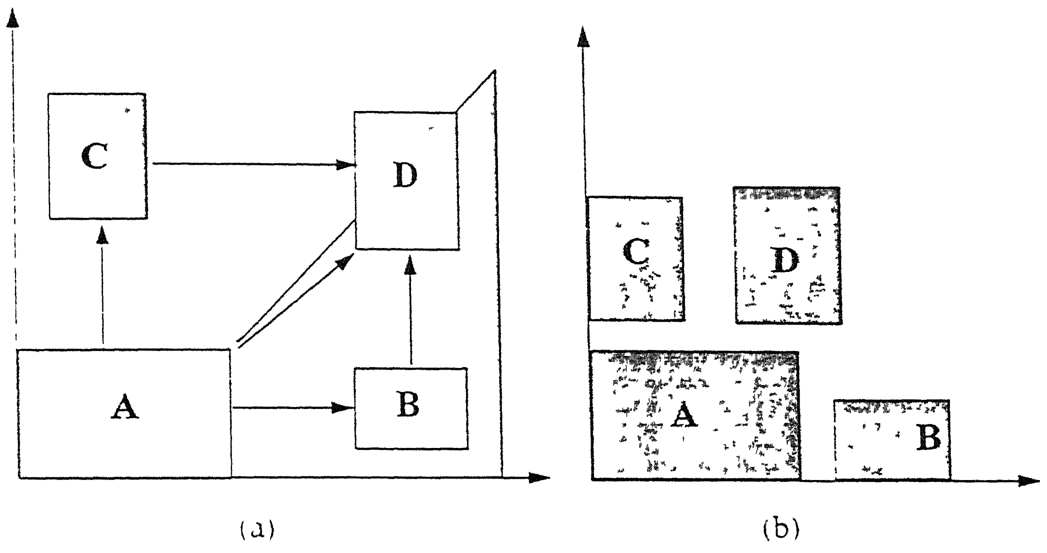


Figure 13 (a) Illustrating concept of ***diagonal shadowing*** Here *A* and *D* are constrained (b) The layout after compaction Since *A* and *D* are constrained, they will not overlap

If a primitive *C* lies in the shadow of another primitive ***B*** which is also in the shadow of the primitive *A* then a spacing constraint from *A* to *C* is redundant even if *C* remains in the shadow of *A* However this statement holds good, if the layout is made up of boxes only In practice the basic layout primitives are not only boxes, but also the polygons The difficulty with the polygons as basic layout entities is illustrated in Figure 14 Here *B* and *C* are in ***shadow*** of *A* and *C* is in the ***shadow*** of *B* however the constraint *(A→C)* is not a redundant one, and actual redundant constraint is ***(B→C).*** Further in the cases of rectangles, if there is a spacing constraint from primitive *A* to primitive *C*, there cannot be a spacing constraint from *C* to *A*, which is not valid in actual cases with polygons and boxes as the basic layout primitives In Figure 14, both the primitives *A* and *C* are related in both ways Keeping all these aspects in view we propose and validate an algorithm, that does not generate any redundant spacing constraint. Instead of shadow being defined by the primitives, we define shadow from every right edge of the primitive while constraining in horizontal direction and from the every top edge of the primitive while constraining in vertical direction Thus we need the following definitions

Shadow. An edge e_2 is said to be in the ***shadow*** of another edge e_1 if it (e_2) is *diagonally visible* from e_1

Front: An edge e_2 is said to be in ***front*** of another edge e_1, in horizontal direction if the following conditions are satisfied

- Both e_1 and e_2 belong to the same primitive
- Edge e_2 is *X visible* from the edge e_1

Left Edge: An edge e_i of polygon ***P*** is said to be a ***left edge***, if the left side of e_i is vacant and ***right side*** of edge e_i is filled to an extent, at least the minimum resolution of the layout Referring to Figure 15(a), the edge $e_i \in P$ can be called a ***left edge*** of ***P***

Right Edge: An edge e_i of polygon ***P*** is said to be a ***right edge*** if the ***right side*** of the edge is vacant and the ***left side*** is filled to an extent at least the minimum resolution of the layout In Figure 15(a) the edges e_{j1}, e_{j2}, e_{j3} are the ***right edges*** of the polygon ***P***

The concept of ***shadow*** and ***front*** is illustrated in Figure 15 In Figure 15 (a), the edges e_i, e_{j1}, e_{j2} and e_{j3} belong to the same primitive and e_{j1}, e_{j2} and e_{j3} are *X visible* from e_i Thus e_{j1}, e_{j2}, and e_{j3} are said to be in ***front*** of e_i. According to the above strategies, in Figure 15(b) edge e_{i2} is in ***front*** of edge e_{i1}, e_{j2} and e_{j3} are in front of edge e_{j1}, and e_{i4} is in ***front*** of e_{i3} The edges e_{j1} is ***diagonally visible*** from the edge e_{i2} So edge e_{j1} can be said to be in the ***shadow*** of e_{i2} Similarly e_{i3} is in the ***shadow*** of e_{i2} and e_{j2} Synonymously ***top edge*** and ***bottom*** edge can be defined to consider shadow in the vertical direction

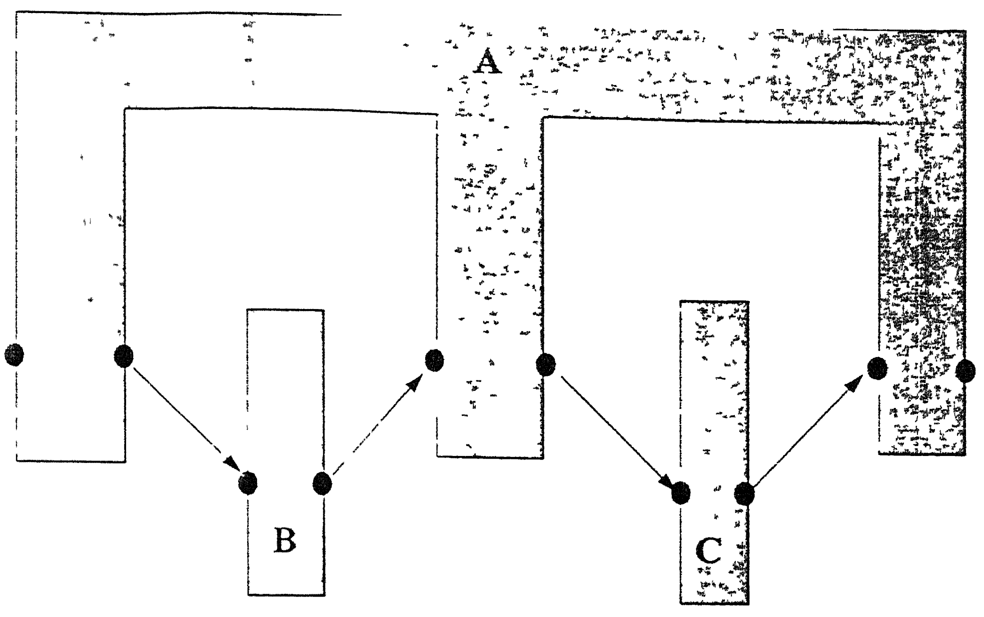


Figure 14 Illustrating the complete set of irredundant spacing constraint that has to be generated for the primitives *A*, *B*, and *C*

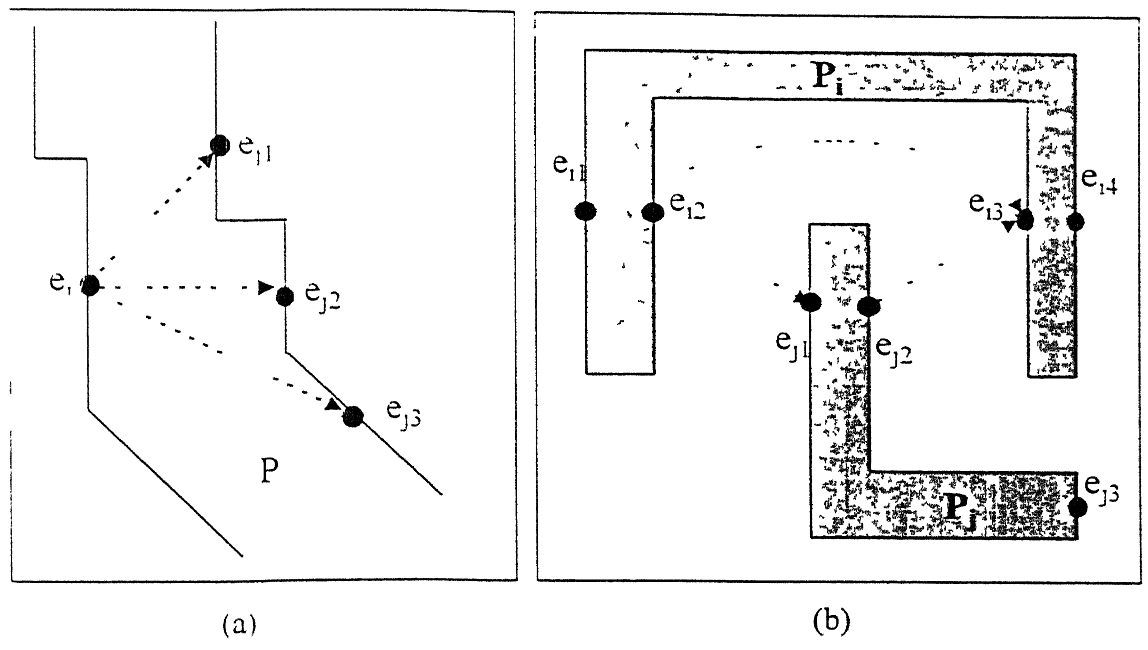


Figure 15 Illustrating the concept of ***Shadow*** and ***Front*** (a) Arrow shows ***Front*** relation (b) Arrow shows the ***Shadow*** relation

2.3.1 Checking for Left and Right Edge

The concepts of left and right edges of an entity are defined earlier To determine whether an edge $e_i \in P_{i,}$ is a ***left edge*** of a primitive, we have to predict a point to the left or right of the edge and check whether that point belongs to the polygon Two subcases are described below Here we predict a point $P(x,y)$ to the right of the edge e_i, and use point enclosure algorithm to check whether this point is inside of the polygon P

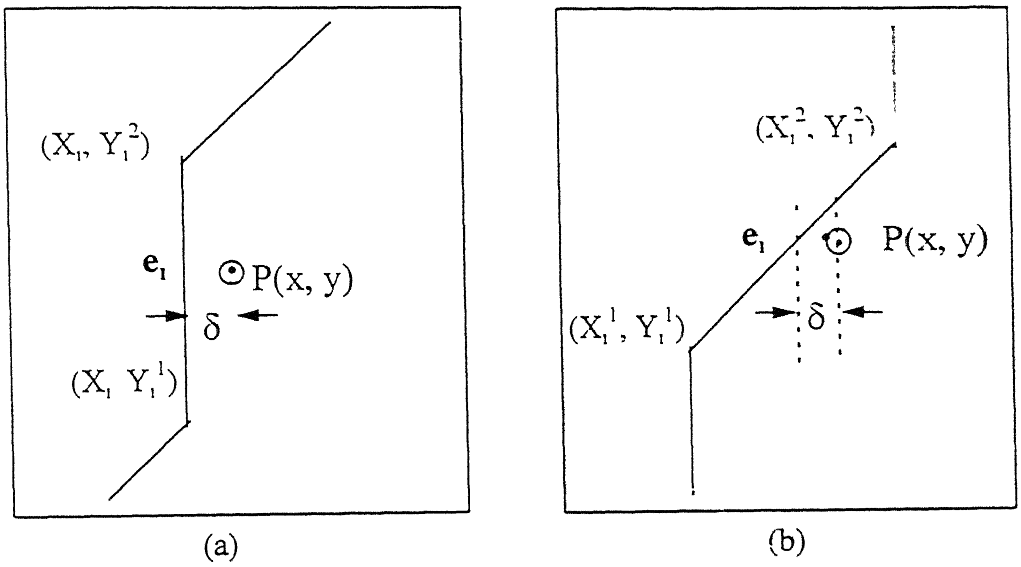


Figure 16 Check for ***left*** and ***right*** edge of a primitive (a) case of a vertical edge, (b) case of an inclined edge

CASE A: e_i is a vertical edge :

Let $P(x,y)$ be a test point sufficiently close to the edge e_i towards its right as shown in Figure 16 (a) The coordinates of the point be

$$X = X_i + \delta$$
$$Y = (Y_i^1 + Y_i^2)/2.$$

where δ is a small real number, smaller than the minimum resolution of the layout

If the test point $P(x,y)$ remains inside the polygon, then the edge e_i must be a ***left edge*** of the polygon, otherwise it is a ***right edge.***

CASE B. e_i is an inclined edge:

This case is shown in Figure 16(b) In this case the coordinates of the test point $P(x,y)$ will be

$$X = (X_i^1 + X_i^2)/2 + \delta,$$
$$Y = (Y_i^1 + Y_i^2)/2.$$

If the test point is inside the polygon, then e_i is a ***left edge***, otherwise it is a ***right edge***

2.3 2 Checking for Front

The concept of ***front*** has been defined earlier Here a method to check whether an edge e_j is in ***front*** of another edge e_i is described Before the edges are checked for being ***front,*** the minimum width type constraints are generated All edges e_j that remains in ***front*** of e_i are constrained by e_i Thus checking for front can be achieved by simply checking for adjacency between the corresponding nodes in the current constraint graph An outline of the procedure ***Front( )*** is given below

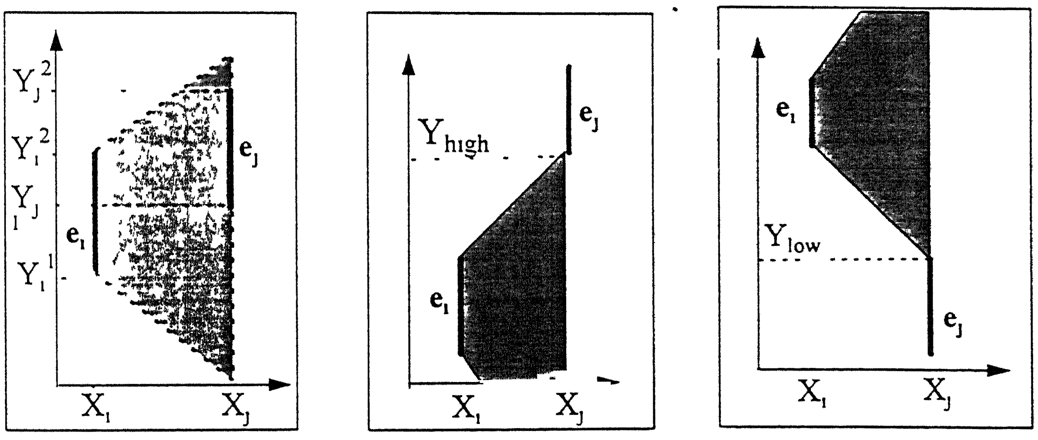
```
Procedure Front($e_i$,$e_j$)
Begin:
    $e_i \in P_i$ and $e_j \in P_j$;
    $v_i$ = the node in constraint graph corresponding to edge $e_i$;
    $v_j$ = the node in constraint graph corresponding to edge $e_i$;
    if ($P_i \neq P_j$) return (FALSE) ;
    if (!Adjacent($v_i$,$v_j$) return(FALSE) ;
    return(TRUE);
End;
```

2.3.3 Checking for Diagonal Visibility

The range of the coordinates in *XY* plane for which an edge e_j is said to be ***diagonally visible*** from another edge e_i is decided by the orientation of the edge in the layout plane Here we will consider only the construction of horizontal constraints and hence, we will restrict ourselves discussing the diagonal visibility in horizontal and diagonal direction only hence we limit the orientation of an edge to be either vertical or inclined only For different orientation of edges e_i and e_j the limits of diagonal visibility is described below In all cases we presume e_i to be a ***right edge*** and e_j to be a ***left edge***

CASE 1. Both e_i and e_j are vertical lines:

Figure 17(a) shows a general case when e_j is ***diagonally*** visible from e_i Figures 17(b) and (c) show the two extreme limits for diagonal visibility We define two variables Y_{high} and Y_{low} that define the two extreme limits for the pair (e_i, e_j) of edges Y_{high} and Y_{low} are determined to be



(a) (b) (c)

Figure 17 Illustrating the ***diagonal visibility*** Case of both vertical edges

$$Y_{high} = Y_i^2 + (X_j - X_i)$$
$$Y_{low} = Y_i^1 - (X_j - X_i)$$

where Y_i^2 is the Y coordinate of the top end point of edge e_i and Y_i^1 is the Y coordinate of bottom end point of the edge e_i. Similarly Y_j^1 and Y_j^2 are defined If $Y_j^1 > Y_{high}$ or $Y_j^2 < Y_{low}$ then e_j does not remains in the ***shadow*** of e_i or e_j is not ***diagonally visible*** from e_i

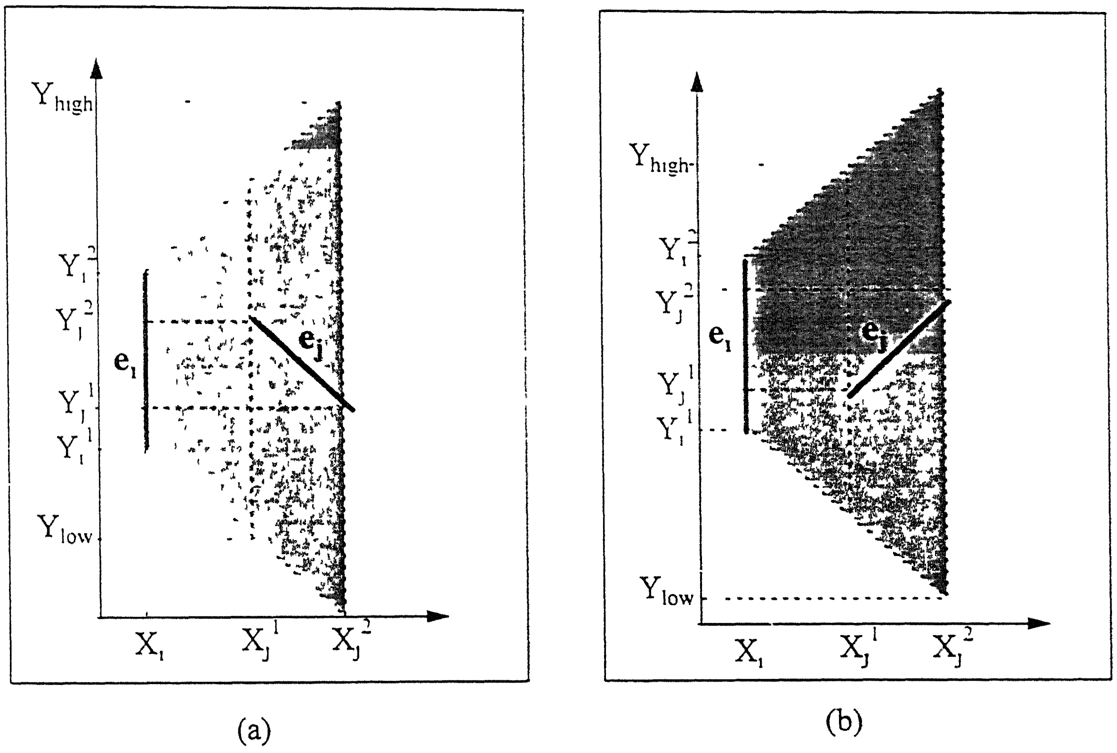


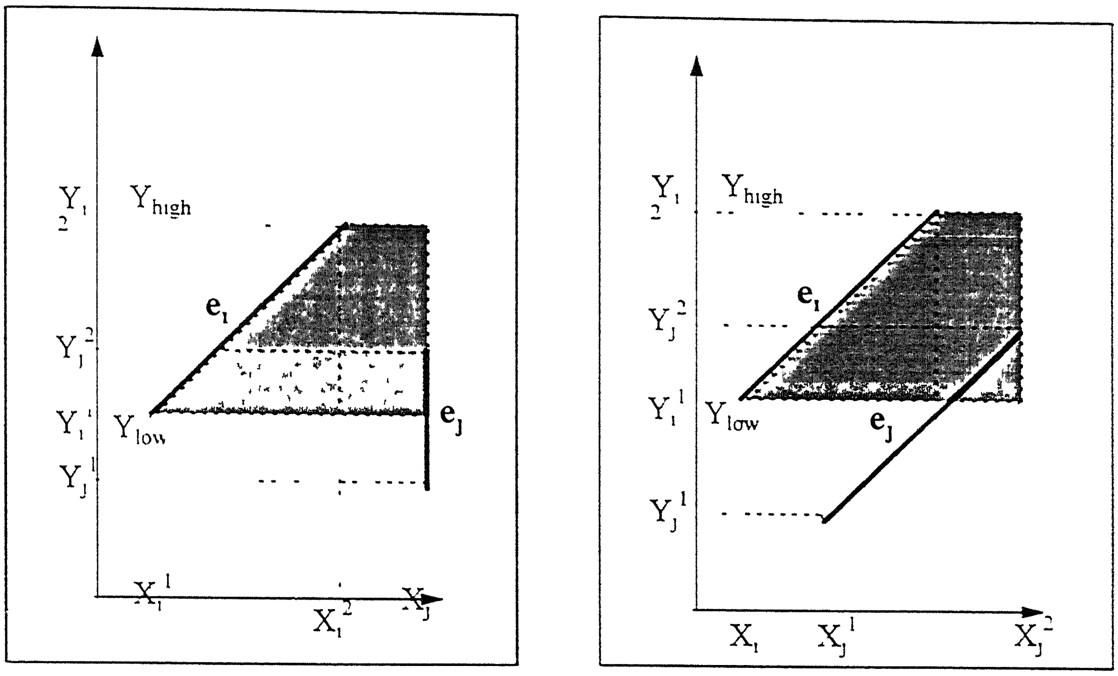
Figure 18 Illustrating ***diagonal visibility***, Case of e_j being an inclined edge

CASE2 : e_i is vertical and e_j is inclined line:

The two sub cases for this type orientation are shown in Figures 18(a) and (b) Let X_1 be the X coordinate of the lower end point of e_j and X_2 be the X coordinate of upper end point of e_j For both the sub cases the extreme values of the Y coordinates are also shown in Figures 18(a) and (b). Y_{high} and Y_{low} take the values:

$$Y_{high} = Y_i^2 + (X_1 - X_i)$$
$$Y_{low} = Y_i^1 - (X_2 - X_i).$$

If $Y_j^1 > Y_{high}$ or $Y_j^2 < Y_{low}$, then e_j is not ***diagonally visible*** from e_i, else e_j remains in the ***shadow*** of e_i.



(a) (b)

Figure 19 Illustrating ***diagonal visibility*** Case of e_i being an inclined edge.

CASE 3: e_i is inclined line, e_j may be vertical or inclined line:

This case is shown in Figure 19 In this case, for simplicity, we restrict the definition of ***diagonal visibility***, and define the shadow considering only the horizontal visibility. We obtain the limits as

$$Y_{high} = Y_i^2 \text{ and}$$
$$Y_{low} = Y_i^1$$

2.3.4 Checking the Shadow

Using the above concepts of ***left edge*** and ***right edge***, and the ***diagonal visibility*** we can formally define the ***shadow*** as follows

Shadow An edge e_j is said to be in the ***shadow*** of another edge e_i if .

- $e_i \in P_i$ and e_i is a ***right edge*** of P_i
- $e_j \in P_j$, and e_j is a ***left edge*** of P_j
- e_j is ***diagonally visible*** from e_i

A formal algorithm to check the ***shadow*** is outlined below

```
Procedure H_Shadow ($e_i$, $e_j$)
Begin:
If ($e_j$ lies to the left of $e_i$) return(FALSE);
If ($e_j$ is a right edge) return (FALSE);
If ($e_i$ is a left edge) return (FALSE);
$Y_{high}$ = Upper limit of shadow region in Y direction for the pair of edges <$e_i$, $e_j$>.
$Y_{low}$ = Lower limit of shadow region in Y direction for the pair of edges <$e_i$, $e_j$>.
If ($Y_{bottom}$ of $e_j > Y_{high}$) return (FALSE);
If ($Y_{top}$ of $e_j < Y_{low}$) return (FALSE);
return (TRUE);
End;
```

2.3 5 Generating the Spacing Constraints

For generating the spacing constraints, we propose a method analogous to ***depth first traversal*** in a ***directed acyclic graph*** [13],[14],[15]. The edges of primitives of the layout resemble the nodes in the ***DAG***, and we traverse along the primitive edges one by one depending on the *front* and the ***shadow*** criteria as the *DAG* is traversed from one node to the other along its arcs The algorithm starts from a ***source vertex (S)*** that represents the left boundary of the layout and visits every edge before stopping The algorithm is cited below

```
Procedure Generate_H_Space_Constraints($G_H$)
Begin:
    Step1:Create two arrays of edge listing called Left[ ] array and Right[ ] array.
    (a) Left array contains all the edges that represent a left edge of a primitive and the right side boundary of the layout (T),i.e. the elements of Left[ ] array can be the destination vertex of a spacing constraint.
    (b) Similarly the Right[ ] array contains all the edges that represent a right edge of a primitive and the left side boundary (S) of the layout, i.e., the elements of Right[ ] can be the source vertex of a spacing constraint.
    (c) Sort both Left[ ] and Right[ ] arrays according to the following strategies:
        • Sort the elements in increasing order of X coordinates.
        • For a set of elements with equal X coordinates, sort in increasing order of $Y_{bottom}$.
        • The first element of Left[ ] array is the edge corresponding to the left boundary (S).
        • The first element of Right[ ] array is the edge corresponding to the right boundary.

    Step 2: Create an array Visited[ ] that maintains the visit status of an edge. Initialize all of its elements to FALSE. Visited[0] to Visited[no_of_left_edges - 1] are attributed for keeping the visit status of edges in Left[ ] array, and Visited[no_of_left_edges] to Visited[no_of_left_edges + no_of_right_edges] keep the visit status of the edges in the Right[ ] array.

    Step 3: root = S;
```

```
      for (each edge $e_i \in$ Left[ ])
       Begin:
             if (!Visited ($e_i$))  H_Traverse (root, $e_i$) ;
        End for;
End;
```

Here *H_Traverse( )* recursive routine for traversing the edge listings, and other procedures such as *Min_space( )* and *Descendant_Visited( )* are outlined below

```
Procedure H_Traverse(p, q)
Begin:
      If (q is a left edge)
      Begin:
            Visited(q) = TRUE;
            Min_space = Minimum required spacing between edges p and q;
            If (p and q fall in interactive layers) join constraint from p to q of value
            equals to Min_space;
            for (each  edge $e_j$ in front of node q) H_Traverse(q, $e_j$) ;
      End if;
      else If (q is a right edge)
      Begin:
             Visited (q) = TRUE;
             for (each edge $e_k \in$ Left[ ] | $e_k$ is in shadow of q)
             Begin:
                  If (!Visited($e_k$)) H_Traverse(q, $e_k$) ;
                  else if(Visited($e_k$) and $e_k$ is not visited by any descendant of  q)
                  Join spacing constraint from q to $e_k$ of weight equals to the
                  minimum spacing between  q and $e_k$.
             End for;
       End if;
```

```
End;

Procedure Min_Space(p,q)
Begin·
        Curr_entry = first entry of design rule table of spacing rules;
        while(all entries are not searches)
        Begin:
            If ((Curr_entry → layer1 = p → layer) and
            (Curr_entry → layer2 = q → layer))
            return (Curr_entry → Spacing_value) ;
            Curr_entry = next entry of the design rule table.
        End while;
        return(-1)
        (-1 corresponds to non interactive layer)
End;

Procedure Descendant_Visited (p,q)
(Returns true if node p is visited from any of the descendant of node q)
Begin :
        If (Adjacent (p,q)) return(TRUE);
        for (each node $n_i$ adjacent to p)
        if (Descendant_Visited($n_i$,q)) return(TRUE);
        return(FALSE);
End;
```

2.4 Generation of Overlap/Extension Constraints

Overlap constraints abstract the overlap/extension layout rules These rules are imposed to preserve the interconnection among the layout elements in different layers/levels Various overlap/extension rules have been described in an earlier section on design rule description In this section we will describe the various algorithms that have been developed as a part of this thesis work to generate these constraints A basic scanline approach is employed to detect the overlap among the layout primitives Special care is taken to generate constraints corresponding to the contacts and vias The following is an illustration of the scanline approach for generating overlap/extension constraint

For generating overlap extension constraints in horizontal direction, i.e. to construct G_H, a horizontal scanline is run in the vertical direction on the layout plane The flattened layout plane information is stored in a common data base, e g , an array containing the vertical edge listings This array is sorted according to a certain strategy such that when the scanline is run, at any instant (or position of the scanline) some selected pairs of edges will generate overlap constraint Whenever a bottom end point is encountered, that edge is inserted to the scanline and when a top end point is encountered, the corresponding edge is deleted from the scanline. The number of distinct steps or positions that a scanline can take is equal to the number of distinct Y-values of the end points At a particular position of the scanline an edge can be inserted or deleted and the deletion takes higher priority than the insertion. From among a set of edges falling on the scanline, a valid overlap is checked according to the following strategies For a particular position of the scanline, let e_i and e_j be the two edges cut by the scanline The various conditions under which an overlap constraint between e_i and e_j is generated are

- If the edges are not previously constrained If these are previously constrained, then it is either by ***minimum width*** type constraint which implies both e_i and e_j are the edges of the same primitive (polygon or box) or it is by a ***spacing*** constraint which implies these edges should be spaced

- If there exists a valid design rule which specifies the required overlap between the layout elements For example in Figure 20, the overlap between e_{i1} and e_{j1} is valid while the overlap between e_{i1} and e_{j2} is not valid as there is no such design rule (*CM* to *CC*)

- If $e_i \in P_i$ and $e_j \in P_j$, P_i and P_j are different primitives and $P_i \cap P_j \neq \phi$, e g , for an overlap constraint to be generated between two edges, the corresponding primitives must overlap, i e one primitive may be completely embedded within the other

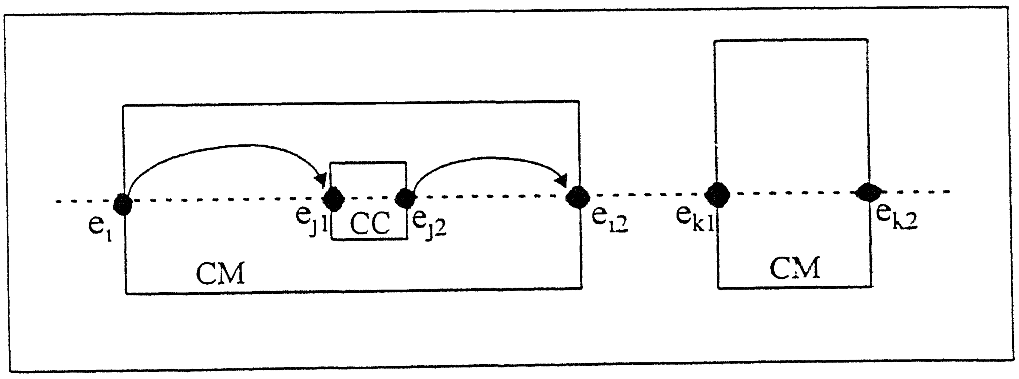


Figure 20 Illustrating existence of valid overlap rule

So at a particular position of the scanline we select all the edge pairs satisfying the above criteria and constrain them The most vital problem that can arise with this approach is the case of contacts and vias as described below:

In Figure 21(a), a contact CC and a metal layer CM is shown. If the scanline approach is used, in this case the following constraints will be generated.

$e_{m8} \rightarrow e_{c1}$ and $e_{c3} \rightarrow e_{m4}$ in G_H and

$e_{m5} \rightarrow e_{c4}$ and $e_{c2} \rightarrow e_{m3}$ in G_V

After the corresponding L P problem is solved, the contact position in the resultant layout will be as shown in Figure 21 (b) At this position it will violate the design rule between e_{c2} and e_{m1}, and between e_{m6} and e_{c8} as there is no such constraint between this pair of edges originally generated

One way to eliminate this problem is to generate constraint between e_{m1} and e_{c2} and between e_{m6} and e_{c4} However with these constraints the wire width will be unnecessarily wider Instead if we generate constraints $(e_{m2} \rightarrow e_{c2})$ and $(e_{m6} \rightarrow e_{c1})$ we prevent the contact position being shifted into the active wire, hence can set the wire width to its minimum width. However these constraints cannot be captured by the conventional scanline approach since the edge e_{m6} leaves the scanline before e_{c1} enters and e_{c1} leaves the scanline before e_{m2} enters So we treat the contacts and vias as a special case and generate the constraints pertinent to the contacts and vias prior to running the scanline

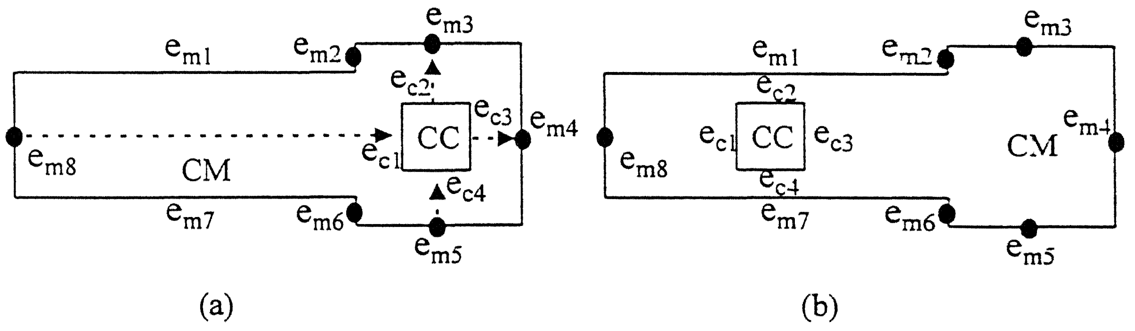


Figure 21 Illustrating shortcoming of scanline approach in generating constraints for contacts and vias (a) Initial contact position. (b) Finally contact is shifted into the wire, violating design rule.

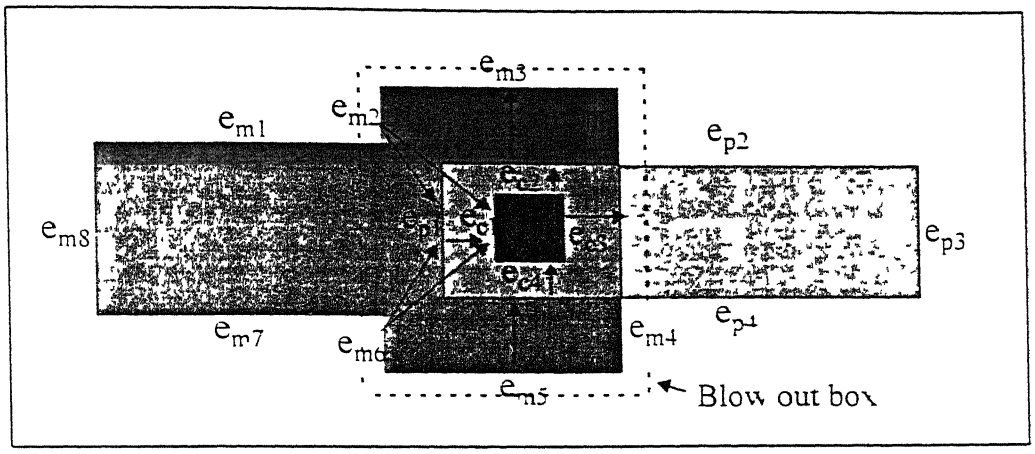


Figure 22 Illustrating constraint generation for contacts and vias ***Blow out box*** is shown in dotted line Edges which are constrained are shown by arrow

For adding the constraints pertinent to the contacts and vias, we take one contact or via at a time We then determine the layout primitives which are actually connected by this contact For example, we have to search for two ***n_diffusion/ p_diffusion/ poly/ metal1*** polygons which encloses that contact. Similarly for a via we have to search a metal1 polygon a ***metal2*** polygon which enclose that via. Once such a triplet, e g , ***CM-CC-CP,*** is detected, we blow out a rectangular region of some arbitrary large length and width surrounding the contact/via. The dimensions of the ***blow out region*** depend upon the minimum required overlap between contact and the characteristic layer This is illustrated in Figure 22 in which a ***metal*** wire and a ***poly*** wire are connected through a contact The corresponding blow out box is shown in dotted line All the edges that fall inside or intersect with the blow out box are taken into consideration for generating constraints We can ensure constraints between ***($e_{m6} \rightarrow e_{p2}$)*** and ***($e_{m2} \rightarrow e_{p2}$)*** by this method.

2.4.1 The Algorithm

```
Procedure Generate_Overlap_Constraint ()
Begin:
    Step 1: Generate Constraints for contacts and vias;

    Step 2: Create an array H[ ] whose elements are the end points of all horizontal and inclined edges in the layout. Create an array V[ ] whose elements are the end points of all the vertical and inclined edges in the layout.

    Step 3 Sort the elements of H[ ] array according to the following strategies:
        • Primary key: increasing order of X coordinates.
        • Amongst a group of points with equal X coordinates those which represent right end points have higher priority.
        • Sort the group of left end points with equal X coordinates in increasing order of Y coordinates.

    Step 4: Sort the elements of V[ ] array according to equivalent strategies

    Step 5: Create an array V_sc[ ] that represents a vertical scanline, whose elements correspond to the points that are currently on the scanline.

    Step 6: Run the scanline from left to right.
    Label 6.1: From H[ ] array, select a group of points that have equal X coordinates.
    for(each point belongs to Group)
    Begin:
        If(point = left end point) Insert that point in the scanline V_sc[ ] at proper position, such that the elements of V_sc[ ] are in the Y sorted order.
    If(point = right end point) Delete the corresponding left end point from
    the scanline V_sc[ ].
End for;

Step 7: for(every point $p_i$ $\in$ V_sc[ ])
Begin:
    for(every point $p_j$ $\in$ H_sc[ ])
    Begin:
        $e_i$ = the edge of the layout corresponding to $p_i$;
        $e_j$ = the edge of the layout corresponding to $p_j$;
        $v_i$ = vertex in in $G_V$ corresponding to $e_i$,
        $v_j$ = vertex in $G_V$ corresponding to $e_j$;
        $P_i$ is the primitive containing $e_i$;
        $P_j$ is the primitive containing $e_j$;
        If (! Adjacent($v_i$,$v_j$))
        If (Valid_Overlap($v_i$,$v_j$))
        If (Min_Overlap($v_i$,$v_j$) $\neq$-1)
        Join constraint between $v_i$ and $v_j$ of weight equals to
        Min_Overlap($v_i$,$v_j$);
    End for;
End for;
If (all groups are not selected) goto Label 6.1:

Step 8: Create an array H_sc[ ] representing the horizontal scanline.

Step 9: Run the horizontal scanline from bottom to the top.
Label 9.1: From V[ ] array, select a group of points with equal Y
coordinates.
for(each point belongs to Group)
Begin:
        If(point = bottom end point) Insert that point in the scanline H_sc[ ] at
        proper position, such that the elements of H_sc[ ] are in the X sorted
        order.
        If(point = top end point) Delete the corresponding left end point from
        the scanline H_sc[ ];
    End for;

    Step 10: for(every point $p_i$ $\in$ H_sc[ ])
    Begin:
        for(every point $p_j$ $\in$ H_sc[ ])
        Begin:
            $e_i$ = the edge of the layout corresponding to $p_i$;
            $e_j$ = the edge of the layout corresponding to $p_j$;
            $v_i$ = vertex in in $G_H$ corresponding to $e_i$;
            $v_j$ = vertex in $G_H$ corresponding to $e_j$;
            $P_i$ is the primitive containing $e_i$;
            $P_j$ is the primitive containing $e_j$;
            If (! Adjacent($v_i$,$v_j$))
            If (Valid_Overlap($v_i$,$v_j$))
            If (Min_Overlap($v_i$,$v_j$) $\neq$-1)
            Join constraint between vi and vj of weight equals to
            Min_Overlap($v_i$,$v_j$) ;
        End for;
    End for;
    If (all groups are not selected) goto Label 9.1:
End;
```

The minimum overlap requirements between various layers are stored in a linked list To find out the minimum overlap between two nodes, the list is searched to find a match of the layers Following is an algorithm for the function ***Min_Overlap( )*** ·

```
Procedure Min_Overlap($v_i$,$v_j$)
Begin·
        layer1 = $v_i \rightarrow$ layer_type;
        layer2 = $v_j \rightarrow$ layer_type;
        If (both $v_i$ and $v_j$ are left edges) layer2 = -layer2;
        else If(both $v_i$ and $v_j$ are right edge) layer1 = -layer2;
        (In other two conditions, i.e,1: $v_i$ is left edge and $v_j$ is a right edge then
        layer1 and layer2 will not be changed. 2: $v_i$ is right edge and $v_j$ left edge
        implies a spacing constraint and will not arise in this case.)
        Search along the design rule list representing the overlap rules for a match
        of layer1 and layer2. If such a match found, return the corresponding
        overlap value of that entry.
        If such a match not found return(-1).
End;
```

2.5 Transistor Resizing

Resizing of all the transistor present in the original layout can be achieved by two different ways The first and easiest approach is to scale all the transistor uniformly setting new length at some user specified value and maintaining the same *W/L* ratio as the previous technology The other approach is to scale the transistors according to user specified values Although the first approach is simpler, the second approach provides the user more flexibility in manipulating transistor sizes for the new technology We provide both the options The various types of transistor configurations that our tool can support are as shown in Figure 23

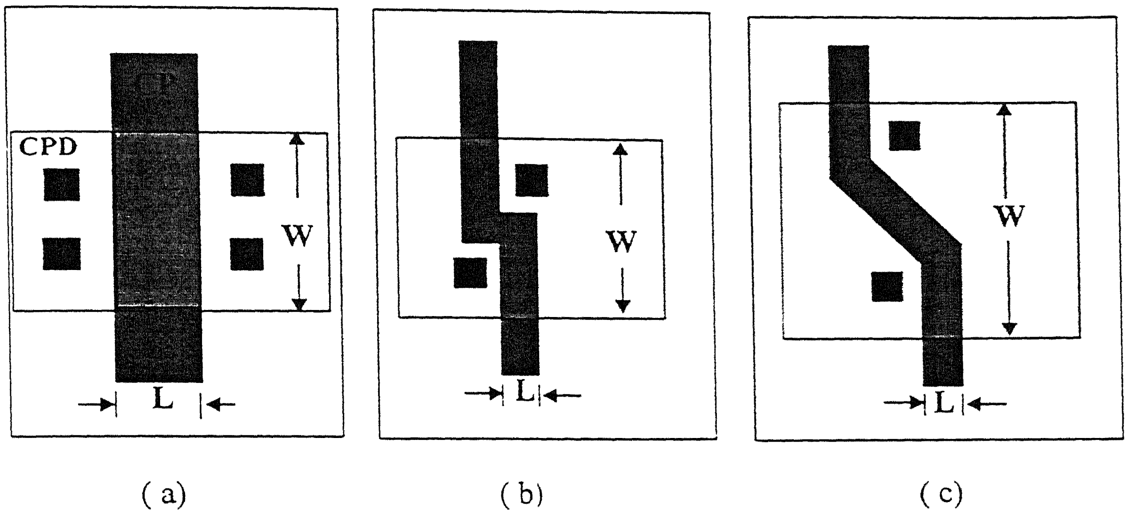


Figure 23 Illustrating various transistor configurations (a) Simple transistor (b) Rectilinear Transistor (c) Transistor with 45^0 inclined feature

The transistor resizing information are stored in a table whose elements are

- A point ***P(x,y)*** in the layout plane, specifying the position of the transistor, to identify a particular transistor This point corresponds to the ***S_W*** corner of the transistor
- ***W/L*** ratio for the transistor in old technology
- Required ***W/L*** ratio of the transistor in new technology

For constant ***W/L*** scaling the only information required from the user side is the minimum channel length of the transistor The channel length and width will be set to a value determined by the ***W/L*** ratio of the transistor in old technology For resizing the transistor according to new ***W/L***, the first task is to detect all the edge pairs which have to be constrained Figure 24 illustrates the various cases that may arise In Figure 24(a), a simple transistor configuration is shown and the required constraint and its weights are shown In Figures 24(b) and (c) the required constraints for other configurations are shown We propose the following formal algorithm to resize the transistors

```
Procedure Resize_Transistor()
Begin:
        for (each transistor in the layout)
        Begin:
            Step1: Find the diffusion polygon Pdiff and the poly polygon Ppoly
            defining the transistor using the point enclosure algorithm.
            Step2: From the polygons Pdiff and Ppoly determine the relevant edges
            that define the actual transistor area.
            Step3: Constrain the appropriate edge pairs according to the new W/L
            values, as shown in Figure 24.
        End for;
End;
```

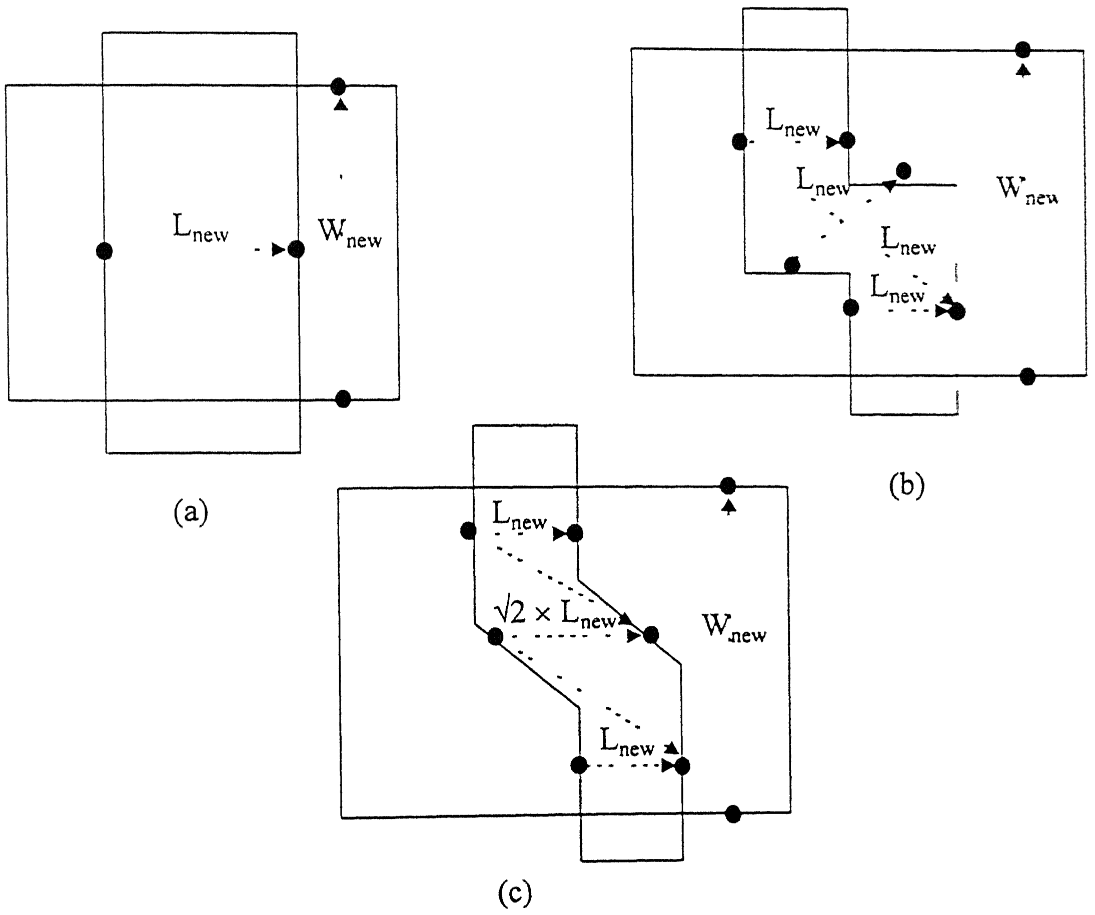


Figure 24 Showing the various required constraints for various transistor configurations

2.6 Other User Specified Layout Constraints

Although all the general types of constraints are discussed, there are still some user specified topological constraints, known as ***user constraints (UC),*** which have to be imposed to provide some flexibility to the user in retargetting the cells The various user constraints that our tool can support are

- Fixing power (V_{DD} and V_{SS}) bus width to a user specified value
- Fixing the connector positions in the layout
- Fixing the cell height

2.6 1 Fixing the Power Bus Width

For any layout the power (V_{DD} and V_{SS}) buses carry the maximum current, hence, these have to be kept wider than the other metal interconnection wires Furthermore, in standard cell based design, the power buses being common to all the cells located in the same row have to be of a fixed width

For fixing the V_{DD} and V_{SS} bus widths, these buses have to be identified at early stages The connectors connecting the V_{DD} and V_{SS} buses are identified in input processing stage This case can be considered as a special case of generating minimum width type constraints, hence can be taken care of at the stage of generating minimum width type constraints In this case a primitive given as an input to the procedure ***Add_Min_Width_Constraint*** is first checked for its membership to the ***metal1*** layer and then is checked for its being a power bus from the information of the power bus connectors If it is a power bus, the weight by which the various nodes corresponding to that entity are constrained is the minimum width of a power bus rather than the minimum width of a metal layer interconnect.

2 6.2 Fixing the Cell Height

For the standard cell based design, all the cells in the same row, have to be of the same height, hence it is essential that the cell height be fixed to a user specified value Cell height can be fixed by explicitly joining a constraint between the two extreme nodes, i e , ***source (Sy)*** and ***sink (Ty),*** of G_V of weight equal to the required cell height

2 6 3 Fixing Connector Positions

Keeping cell abutment in view, there must be a common platform for positioning the I/O pins of a cells For this reason the connectors of a cell have to be fixed at some position along the cell boundary A standard rule is to fix the connectors at a distance of ***(integer + 1/2)*** of the ***routing pitch***, from the origin of the cell ***(S_W*** corner of ***Abutment box***) along the cell boundary All the cells designed according to the above strategy have their connector positions fixed along the cell boundary, hence can be abutted easily The connector positions can be fixed by identifying the corresponding mask polygons of ***metal1/metal2*** layer, the connector is connected to, and joining explicitly constraints from ***source*** to the respective polygon edge of the value equals to the distance of the connector from the ***source***

Chapter 3

Solving the Constraint Graph

3.1 Introduction

In Chapter 2 we discussed the process of generating constraint graph, in which the layout problem is formulated as a linear programming (L P) problem The L P problem consists of a set of constraints and an objective function The aim is to solve this problem in order to optimize the objective function, keeping in view that all the constraints are satisfied Furthermore, in our case each equation in the L P problem has at most two variables Hence the problem in algebraic form is translated into a graph theoretic form The following is an illustration

Let the set of constraints be as follows

$$Xb - Xa \geq 8 \text{ ——— } 1$$

$$Xc - Xa \geq 12 \text{ ——— } 2$$

$$Xc - Xb \geq 10 \text{ ——— } 3$$

$$Xd - Xb \geq 10 \text{ ——— } 4$$

$$Xd - Xc \geq 5 \text{ ——— } 5$$

$$Xe - Xc \geq 10 \text{ ——— } 6$$

$$Xe - Xd \geq 2 \text{ ——— } 7$$

These set of constraints are represented by a directed graph as shown in Figure 25

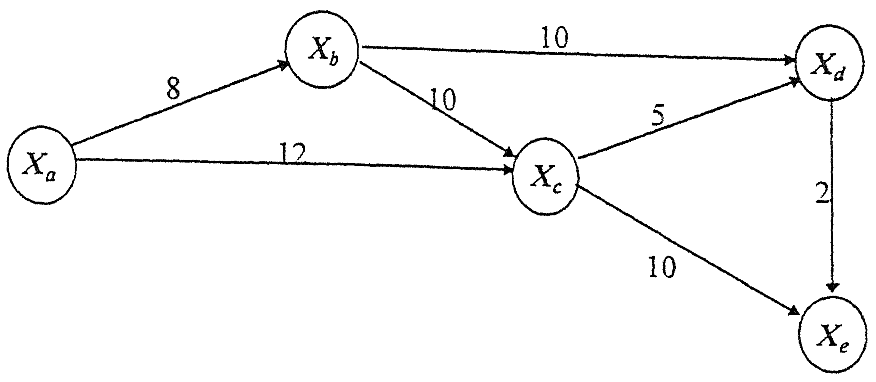


Figure 25 Illustrating graph representation of a set of linear equations

The task at hand is to select a group of constraints that leads to a solution to the problem Among the set of constraints some may be redundant, e g , constraints 2, 4 and 6 in Figure 25 are redundant constraints Our task is to detect and remove all such redundant constraints present in the constraint graph

It is possible that there may be several groups of constraints and that each group satisfies all the design rules There arises the problem of choosing a group that minimizes certain parameters specific to the layout, which is called optimization Each group of constraints leads to a different basic feasible solution. We follow the simplex method for solving the linear programming problem, but the problem is actually solved in graph domain rather in conventional tabular manner

The key idea of simplex method in solving the linear programming problem is to select a ***basic feasible solution (bfs).*** This initial bfs may not correspond to an optimal solution From this bfs, successively, one or more constraints are removed and equal number of constraints are added without violating any design rule, and in a manner to reduce the objective function value In this chapter we discuss the algorithms and various relevant issues related to finding a basic feasible solution to the problem, formulation the objective function and finally we will discuss algorithms and issues related to the optimization of a layout

3.2 Basic Feasible Solution

Formally linear programming problem can be defined as follows

Instance : A real $m \times n$ constraint matrix A, a real m-vector b, and a real n-vector **c**

Configuration: All real n-vectors

Solution: All n-vectors x that satisfy the linear constraints $A \quad x \geq 0$

Minimize $C^T x$ where C = cost vector

The sets of feasible solutions to the constraints when represented in $\mathbf{R}^n$ plane, define a region (may be bounded or unbounded) called a Polytope Every point inside the polytope is a solution to the problem The basic task of optimization is to find one among those points that minimize the cost function Although every point inside the polytope represents a feasible solution, the following lemma restricts the search region of the polytope and minimizes the number of points to be checked for obtaining the minimum cost function

Lemma :

Let ***P*** be the polytope defined by the set of inequality constraints and $c(x) = C^T x$ be the linear cost function Then the minimum cost on ***P*** with respect to $c(x)$ is attained in a corner of ***P***

So instead of checking for the objective function value at any arbitrary point inside the convex polytope, the check is restricted only to the corner points of the polytope When represented in graph domain each of the corner points of the polytope corresponds ro a spanning tree Each basic variable corresponds to a node in the constraint graph (also in the spanning tree) Each inequality (facet of the polytope) corresponds to a constraint in the constraint graph Conventional approach of solving linear programming utilizes a tabular format to store all the information But the layout problem consists of a large number of variables for which the tabular format for storing and solving is quite space expensive Conventional approach for solving L P problem by simplex method is described in [16] Here, we describe only the algorithms and methods to solve the L P problem using the graph theoretic approach

Conceptually our basic objective is to obtain a layout of minimum overall area To achieve this the circuit elements and vertical interconnections are moved horizontally to the leftmost achievable position Similarly the horizontal interconnections are moved to the bottommost possible location Presently we generate vertical and horizontal constraint graphs taking all the lower bound type constraints into consideration, i e all the constraints are of the form

$$X_b - X_a \geq Wt.$$

where Wt is a positive real number specifying the required minimum separation between the two nodes Thus our constraint graph G is in fact a single source single sink, edge weighted directed acyclic graph

We define a solution set of G as the set $\{ V_i(x), 0 \leq i \leq n \}$ such that it satisfies all the inequality constraints implied by the edges of G Since V_0 represents the source (S_v), e g , the left boundary of the layout, its position is normalized to zero A basic feasible solution set $\{ V_i(x) \}$ of G is defined as the minimum solution set, if, for any other solution set $\{V_i'(x)\}$ we have $V_i(x) \leq V_i'(x)$ All the nodes are placed at as minimal distance as possible from the source node The value of $V_n(x)$, e g , distance of sink vertex is the value of minimum possible width of the compacted layout. As the primary goal is to minimize overall area of the layout, all the nodes that fall in the longest path from the source to the sink remain invariant to the optimization process. The only nodes that can be moved during optimization phase are those which do not fall on the longest path from the source to the sink and which provide some slacks for movement

3.2.1 Removing Redundancy from the Constraint Graphs

The constraint graph so far constructed may have a number of redundant constraints Some of these redundant constraints can be removed before determining a bfs to the problem The removal of redundant constraints helps to speed up the subsequent optimization steps To remove all redundant constraints requires the following graph reduction For each constraint $X_j - X_i \geq W_{ij}$ the longest path length from V_i to V_j is calculated If the longest path length is larger than W_{ij} then the constraint is redundant and should be removed. The task of graph reduction is equivalent to the construction of the transitive closure of the graph, which is though polynomial in time is still too expensive $O(N^3)$ Instead we use the simpler task of removing those arcs which satisfy triangle inequality Let us consider three vertices V_i, V_j V_k which are constrained by weights W_{ij}, W_{kj}, and W_{ik} as shown in the Figure 26 We will remove the arc $(V_i \rightarrow V_j)$ if the triangle inequality $W_{ij} \leq W_{ik} + W_{kj}$ holds good Following is a formal algorithm to trim some of the redundant arcs from the constraint graph G

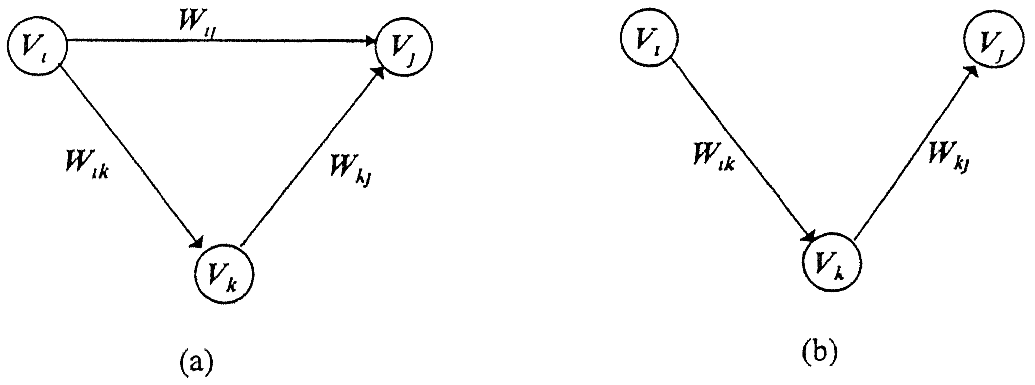


Figure 26 Illustrating removal of redundant constraint Here $W_{ij} \leq W_{ik} + W_{kj}$.

```
Procedure Arc_trimming
Begin
    for (each arc $e_{ij}$ = (Vi,Vj) belongs to G) do
    Begin
        for (each arc $e_{ik}$ = (Vi,Vk) belongs to G) do
        Begin
            if (arc $e_{kj}$ = (Vk,Vj) belongs to G exists and
            $Wij \leq Wik + Wkj$)
            Remove arc $e_{ij}$ (Vi,Vj) from the graph G;
        End;
    End;
End;
```

The number of possible arcs going out from a node V_i or V_k is on an average, a constant, independent of the size of N, since most of the arcs are between neighboring nodes The average time complexity of the arc trimming algorithm is some constant times $|E|$ The worst case complexity can be as bad as that of finding the transitive closure

3.2.2 Searching a Basic Feasible Solution

After a selected set of constraints is removed from the constraint graph, we solve the longest path problem in order to find a basic feasible solution. The following is an algorithm for determining the bfs to the problem

```
Procedure Search_bfs
Begin
    1. From the graph G select a spanning tree T.
    2. Compute the distances of all nodes of T.
```

```
        3. Check whether the tree T thus selected leads to a feasible solution.
        4. If the tree is a feasible tree return.
           else
                Make an elementary tree transformation.
        goto 2.
End
```

The above procedure starts from an arbitrary spanning tree selected from the constraint graph The tree thus selected may not satisfy all the design rules The edges of G which are to be added to the spanning tree in order to make it feasible are detected and equal no of edges are deleted from the spanning tree by a process called tree transformation The tree transformations are repeated until there are no infeasible arc present in the resultant tree This procedure is applicable to both the horizontal and vertical constraint graphs to determine a bfs The steps defined in the above algorithm are explained and the algorithms for these steps are described in the following section.

3 2 2 1 Selecting Spanning Tree from the Graph

The spanning tree problem is basically an edge selection problem More precisely, given an edge weighted graph $G = (V,E)$, the problem is to select a subset of edges $E' \subseteq E$ such that E' induces a tree $T = (V,E')$ that spans every vertex of the set V The algorithm to select such a spanning tree is described as follows

Let G be a directed graph, free of selfloops (in fact the constraint graphs for the layout problems are self loop free) Let G consists of n vertices and e edges. The vertices are labeled $1,2,\ldots n$ and the graph is represented as a linked list. For selecting a spanning tree, the edges of the graph are selected one by one and inserted into the tree if it does not produce any loop in the resulting tree Each vertex is assigned to a set Thus at start there are n different sets At any stage each set represents a partial spanning subtree constructed so far For a connected graph *G*, at the end, all the sets are merged into a single set which contains all the vertices and a selected no of edges defining the spanning tree

Lemma :

In the constraint graph *G*, there is at least one directed path from the source node to every other node There is at least one directed path from every node to the sink node

In the process of selecting the spanning tree, the root of the tree corresponds to the source node and one of the leaf node corresponds to the sink node We are required to select a subset of edges $E' \subseteq E$ such that there is one and only one directed path existing from the source node to every other node This implies that every node in the spanning tree must have one and only one in degree, the in degree of the source node being zero

Define two arrays ***D[ ]*** and ***S[ ]*** as

For a node ***u*** belonging to the spanning tree, ***D[u] = TRUE*** if the ***in_degree*** of ***u*** is not equal to zero This implies the node ***u*** remains at the destination of some edge in the spanning tree Similarly for a node ***u*** belonging to the spanning tree, ***S[u] = TRUE*** if the ***out_degree*** of node ***u*** is not equal to zero That means the node ***u*** is the Source of some edge belonging to the tree If at any stage, an edge ***(u,v)*** is inserted to the spanning tree formed so far, ***D[v]*** becomes TRUE and it will prevent any other edge ***(u',v)*** to enter to the spanning tree Thus all the edges for which ***D[v]*** is TRUE, are rejected from the spanning tree and these are marked as co_tree edges If ***D[v]*** is FALSE, an edge ***(u',v)*** can be inserted to the spanning tree. But the exact method of insertion depends upon the value of ***D[u], S[v], and S[u]*** Figure 27 is an illustration An algorithm to select a spanning tree from the graph is given below

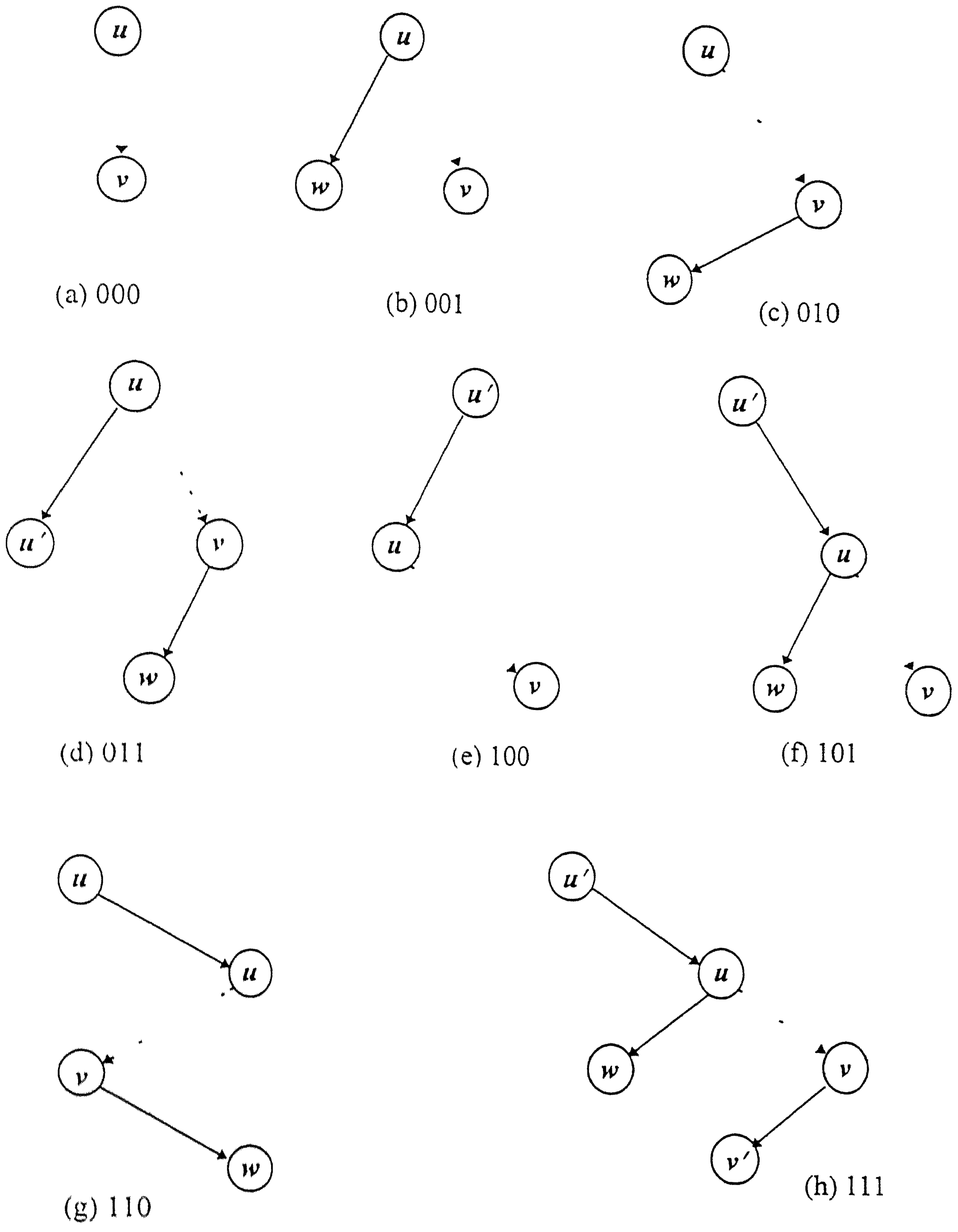


Figure 27. Trees arranged in order of ***D[u], S[v],*** and ***S[u].*** Dotted edge ***(u,v)*** represents edge to be inserted to the trees Solid edges represent the edges of the trees formed so far

```
Procedure Select_Spanning_Tree(G)
Begin
 1. Create a tree node and assign its various parameters corresponding
    to the source node (S_v) of graph G
 2. for (all node u) Initialize :
                      D[u] = FALSE;
                      S[u] = FALSE,
 3. for (each edge (u,v) ∈ G) do
    Begin:
        if(D[v] = TRUE) Begin ·
              Mark edge e (u,v) as co_tree edge ;
              Continue ;
              End if ;
        Check D[u], S[v], and S[u] ;
        Case 000 :
        Neither u nor v is included in any of the subtrees formed so far. Create two
        tree nodes u and v. Assign various parameters to u and v according to the
        corresponding node in graph G. Join an edge from u to v and mark v as the
        son of u.

        Case 001 :
        Node u already exists in one subtree while node v is not in any of the
        subtrees. Create a tree node v, make suitable mapping between node v and
        corresponding graph node. Link u and v appropriately.

        Case 010 :
        Node u does not exists in any of the subtrees, while node v exists in one of
        the subtrees. Create a tree node u and make suitable mapping between u and
        corresponding graph node. Join an edge between u and v such that v will be
        recognized as a son of u
        Case 011 .
        Both the nodes u and v are present in separate subtrees Tu and Tv. Link u and v appropriately such that the two subtrees Tu and Tv will be merged into a single tree.

        Case 100:
        In some subtree, there is an edge from some node u' to node u and node v does not exist. Create a tree node v. Join an edge between u and v to make v the son of u.

        Case 101
        Node u is in one of the already constructed tree while node v does not exist. Create a tree node v and link u and v appropriately.

        Case 110 :
        Nodes u and v exist in two separate subtrees Tu and Tv. Join an edge from u to v so that v will be son of u and the two subtrees will be merged into a single tree.

        Case 111 :
        Nodes u and v exist in two separate subtrees Tu and Tv. Link u and v appropriately to merge the two subtrees into a single tree.

    End for ;
    End ;
```

3 2 2 2 Computing Distance of Every Node of the Spanning Tree

After selecting a spanning tree from the constraint graph, the next task is to compute the location of various nodes Since the information about the spanning tree is stored in binary tree form, computing distance can be achieved by a ***pre-order*** traversal of the binary tree The source node (S_v) is fixed at the location '0' The location of successive nodes are determined by adding the edge weight to the distance of their father node For an example, let X_u be the position of node u, edge $e_{uv}(u,v)$ belongs to tree T and W_t = weight of edge $e_{uv}(u,v)$. Then the position of node v is given by $X_v = X_u + W_t$ The following is a recursive routine that traverses the tree in ***pre-order*** and determines the position of all nodes

```
Procedure Compute_dist(T)
Begin:
      if(T != NULL)
      Begin :
            if(T→father == NULL) dist(T) = 0;
            else
            Dist(T) = Dist(T →father) + edge weight;
            Compute_distance(T →son) ;
            Compute_distance(T → brother) ;
      End if ;
End ;
```

3 2 2 3 Determining Feasibility of Tree

The spanning tree selected so far may not lead to feasible solution, that satisfies all the design rules, i e , some of the constraints originally present in the constraint graph may not be satisfied The following lemma suggests the condition under which the selected spanning tree represents a basic feasible solution.

L mma Every edge $(u\ v) \notin T$ defines one fundamental circuit/loop in the tree T D fine length of a fundamental loop defined by an arc $(u\ v) \in T$ as

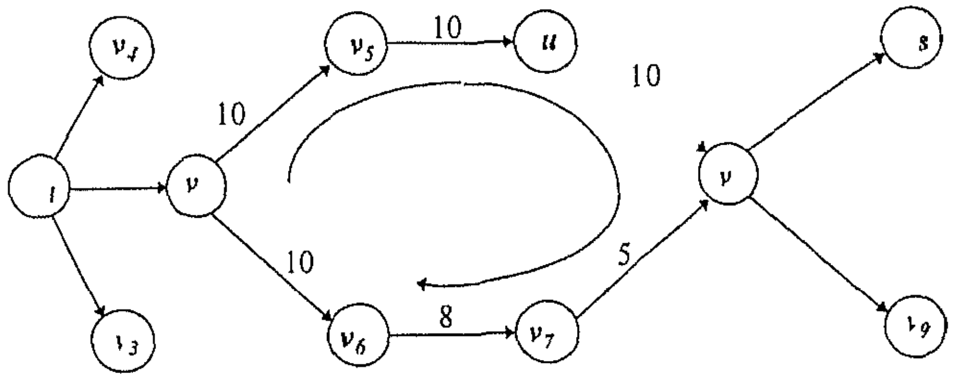
$L^{fl} = \sum Wt(v_i\ v_j) \times Dir(v_i\ v_j)$ for all $_i(v\ v_j)$ belong ng to the fundame t l loop \ her $c_j = (v_i\ v_j)$ is an arc in the fundament l loop defined bv the ed e $(\iota\ )$
$Dir(u\ v) = +1$ if the edge $(v_i\ v_j)$ s in same direction as the direction of arc $(\iota\ v)$ lse $Dir(u\ v) = 1$

The corresponding spanning tree T leads to a fe sible solution if for all fundamental loops th length of the fu damental loop is non positi e In a formal wa\ we sav a tree T is feasible if for all cotree edges $(u\ v) \in T$ $L^{fl} \leq 0$

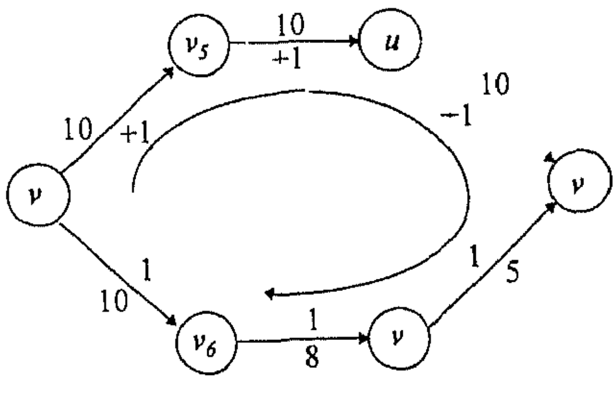
The following is an explanation of the above lemma

Let us consider the spanning tree as sho\n in F gu e ?8 The dotted arc represents the edge of the cotree and the solid a cs represent the ed es of th tr e Ed e weights are spe ified along with the edges Consider an edge $(u\ \iota)$ of the cotree The fundamental loop defined bv the edge $(u\ v)$ is $u \rightarrow v \leftarrow v_7 \leftarrow v_6 \leftarrow v\ \ \rightarrow v_5 \rightarrow u$ The direction of edges a shown in Figure ?8(b) The length of the fundamental loop = 10 10 + 10 5 8 10 = 7 which implies that the original tree is not a feasible one The explanation for this infeasibility is as follo vs

The node v is not placed at its minimal possible position it is placed at 23 units awav from node v_2 but if ve follow the path $v\ \ \rightarrow v_5 \rightarrow u \rightarrow v$ we will obtain the position of node v as 30 units from node v which is the min mal (left most) position of node v If we restrict ourselves to the original spanning tree it will iolate the design rule between the node u and node v These two nodes should be separated by a minimum distance of 10 units while these are actually separated by onl\ units only To make the solution feasible we have to delete the edge $(v\ \ v)$ from the original spannın tree and add edge *(u v)* to the tree This ıs called an ***element ry tree tr nsformation*** ın which one spannıng tree ıs extracted from anoth r on by emo ıng one edge from the ınıtıal spannıng tree and then addıng another ed e to ıt Here e wıll dıscuss the algorıthm to check whether a gı en spannın tree leads to feasıble solutıon If the tree T leads to an ınfeasıble solutıon we vıll det ct the exact d *(u v)* $\in T$ whıch should be added to the spannıng tree to make the solutıon feasıble



(a)



(b)

Fıgure 28 Illustratıon of a tree leadıng to a feasıble solutıon

```
Procedure Check_Feas bılty(T)
B   ın
        for (e ery   d e (u v) ∈ T ) do
        B   ın
            ı t(u v) = weıght of arc(u   v)
            $L^{fl} = wt(u\ v) + Dist(u)\quad Dist(v)$
            ıf($L^{fl} > 0$)
            Be ın
                Infeasıble arc = (u v)
                r turn(FALSE)
            Lnd ıf
        End for
return(TRUE)
End
```

There are E   N +1 edges present ın cotree Hence the tıme complexıty of the al orıthm ıs O(E)

3 2 4 Tree Transformatıon

In the prevıous stage we detected the edge *(u v)* whıch must be added to the spannıng tree ın order to make the solutıon feasıble Sımultaneously one of the edge whıch ıs orıgınally present ın the spannıng tree must go out of the tree Any edge e_{ij} of th fundamental loop that belongs to the *T* and wıth dırectıon +1 may be a suıtable candıdate for removal Wıthout loss of generalıty we choose the edge *(u v)* ∈ ***T*** to be removed Followıng ıs the algorıthm for tree transformatıon

```
Procedure Tree_transformatıon(T u v)
Begın
```

```
    1 Detect the edg  (u  v)  ∈ T
    2 Mark  dσe (u  v) as co_tree ed  e
    3 Mark ɛdσe (u v) as tree edge
    4 Mak  suitable modification in the ori  inal tree T to remove  d  e
      (u v) and to add the ed  ɛ (u  v)
End
```

3 3 Optimization

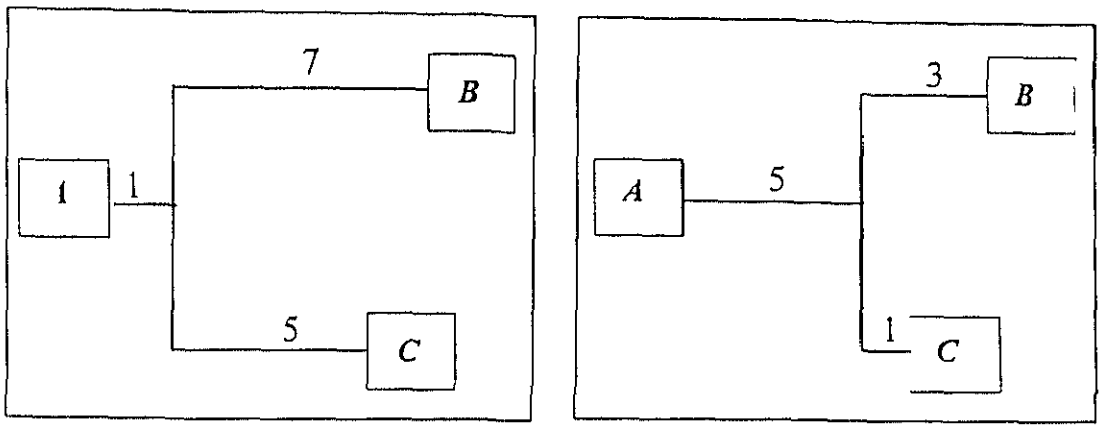
3 ɔ 1 Need of Optimization

The basic feasible solution obtained i previous st p shrinks every elem nt to a corner (origin) of the layout plane and produces a layout of minimum overall area The overall area of the initial layout does not depend on all the primiti es/elements only a few control the layout area We call those elements as the ritical elements specific to a given layout instance However the noncritical elements have plenty of freedom to position themselves in the original layout within som l m ted ran e without affecting the overall layout area One of the important considerations n mi rating the layouts is to preserve the uniform distribution of la out elements throughout the entire layout plane The layout obtained from the solution of the longest path problem suffers from the following demerits

All the non critical elements are packed towards one orn r of the la out which introduces higher degree of non uniformity to the layout

The resultant layout has a longer total wire length Hence it contributes to more parasitic resistance which degrades the circuit performance

Figures 29 and 0 illustrate the importance of wire length m n mization In Fi u e 29(a) the blocks *A* *B* and *C* are interconnected by vires and the total \ı e len th amounts to 1 units This configuration is obtained afte sol in the lon est path problem and without wire length optimization Figure 29(b) is the corresponding la out obtained after wirelength optimization Here net wire l ngth = 9 units



(a) (b)

Fi ure 9 Illustrating wire length minimization

In Figure 0(a) one end of the wire W is pulled to the left most ach evable position while it s other end is connected to an element on the critical path Hence the net wire len th is unnecessarily higher In Figure 0(b) th same layout is shown after optimizing the wire length Here the net wire length is reduced and hence results in an improved circuit performance without compromising the net layout area In Figures 31 and 32 we present more cases where circuit performance can be improved by optimizing the layout Consider a contact location where two wires belonging to different lavers are interconnected together as shown in Figure 31 Here *wire1* is m de up of *metal1* and *wire2* is made up of *polysillicon* Resistivity of polv is orders of magnitude higher than that of metal Hence wires on polv layer significantly increases si nal delay In Figure 31(a) the contact is placed at its minimal possible position However the contact has some freedom in positive X direction In Figure 31(b) the same layout is shown with the same net wire length but with the contact shitted tow1rds right This results in a decrease in vire length on *pol}* layer while the vire length on the *metal1* layer increases by an equal amount B} this proc ss the ov rall length of the wire remains the same as its previous value but the effective r sistance is reduced by significant amount vhich ensures bett r circuit pe formance

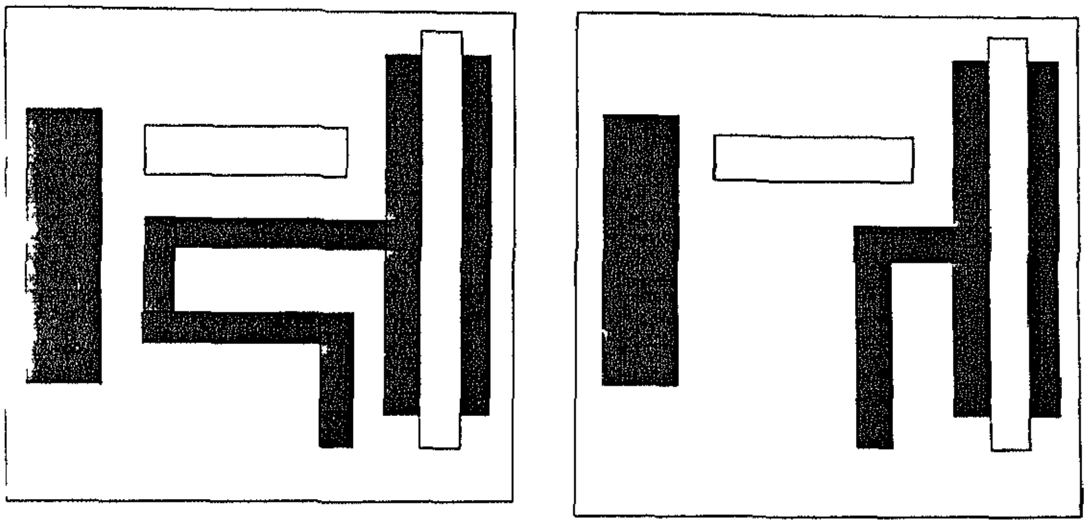


Figure 30 (a) A layout compacted without wire length minimization
(b) The same layout compacted with wire length min mization

In standard cell based design the I/O sp cifications hence the connector positions are fixed along the cell boundary Figure 32 shows a cell for which the connector position is fixed Consider the case where a *metal2* w e associated with a connector making contact with a *metal1* wire (Figure 32(a)) In Figure 32(a) the lavout is obtained after placing all the elements at their minimal possible positions vhich results in extending the *metal2* wire and shrinking the *metal1* wire If we assume *metal2* layer is more costly than *metal1* layer then the lavout involves mo e cost F ure 3?(b) shows an alternative arrangement for the same functionalit but vith shorter wire length for *metal2* layer This arrangement is a result of wirelength optimization and involves lesser cost

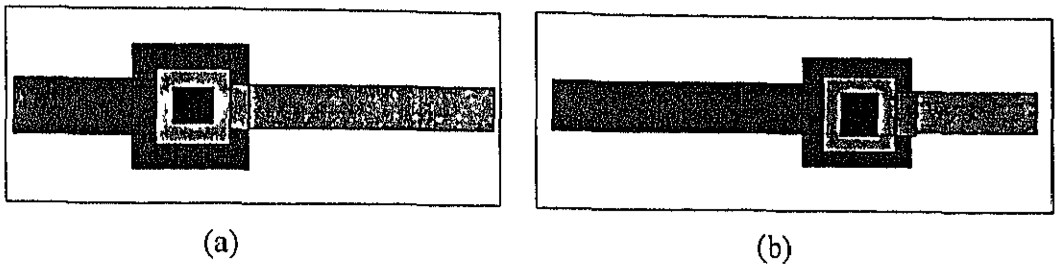


Figure 31 Moving the contact to reduce overall parasitic resistance
(a) Layout without wire length optimization involves higher resistance
(b) Layout after wire length optimization Length of wire on *poly* layer is decreased expanding wire on *metal* layer Overall length remains same but effective resistance reduces significantly

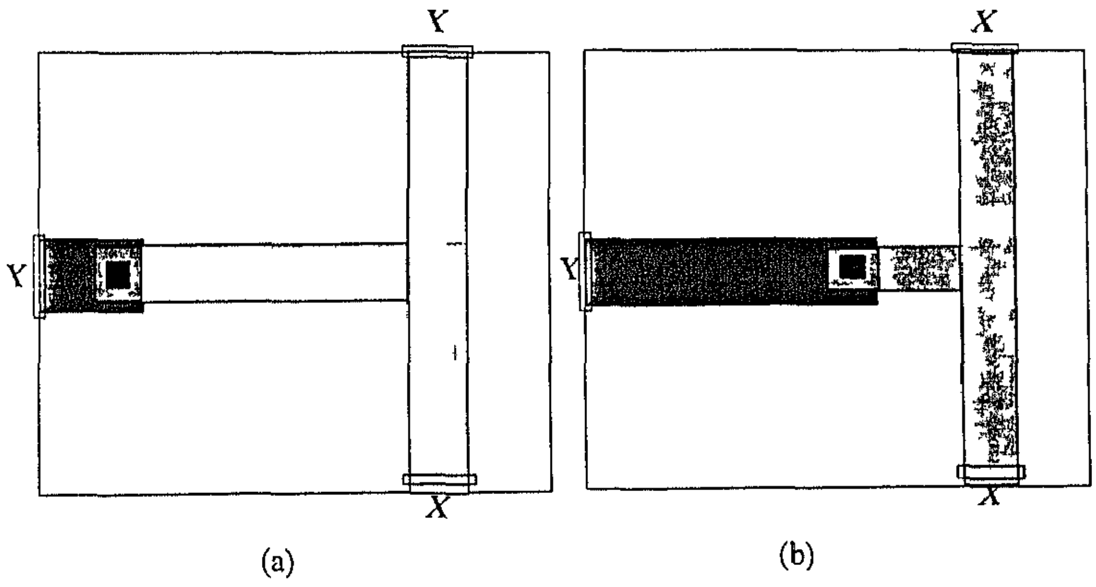


Figure 32 Illustrating wire length optimization X represents connector position
(a) Layout before optimization involves more wire length on ***metal2*** (more costly) layer Hence involves more cost (b) Layout after wire length optimization Wire length in ***metal2*** layer is reduced by increasing wire length on ***metal1*** layer Overall resistance is reduced

3 3 2 Formulating the Cost Function

The characteristic parameter specific to a layer type which is of our interest is the sheet resistance of the layer The sheet resistance of various layers are given in Table 4 Chapter 4 Various ways of formulating the cost function are described in [17] and [18] In [18] two different strategies have been considered to model the cost function One of them involves a relation with the perimeter of the mask polygon to model the cost function while the other takes into account the area of the mask polygon to model the cost function In our approach we associate each edge with a cost parameter which we define as given below

Definition

For an arc *(u v)* associated with a particular layer the cost of arc C is defined by the rate of change of resistance of the associated mask layer with respect to the length of the arc If the arc is not associated with a single layer or not at all associated with a layer its cost is set to zero

Let for an arc *(u v)* R = Resistance of the corresponding mask considering the direction *(u→v)* as the length of the sheet and the other dimension as the width of the sheet Let R be the sheet resistance of the corresponding mask layer

L = Length of arc *(u v)*

W = Width of the sheet (refer to Figure 33)

Cost $C = dR / dL$

$R = R \times$ Number of squares on the sheet defined by arc*(u v)*

$= R \times (L / W)$

Thus Cost $C = R / W$

As the dıstance between u and v changes whıle determının the feasıble solutıon len th and wıdth of the sheet should refer to the dımensıons of the most recently avaılable sheet So we determıne the cost parameters after ettıng a feasıble solutıon to the problem The cost functıon to be optımızed can be formulated as

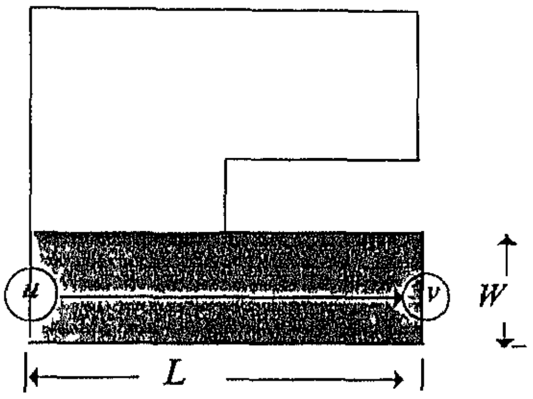


Figure 33 Thıs polygon represents a partıcular mask layer Cost of arc $(u\ )$ wıll be decıded by the ıncremental resıstance of the shaded regıon

$$Z = \sum_{u\ v \in G_H} C \quad \times (X \quad X_j) + \sum_{u\ v \in G_V} C \quad \times (Y \quad Y_j)$$

The two dıfferent terms are optımızed separately ın two passes one ın horızontal and the other ın vertıcal dırectıon

We have assıgned a cost to each of the arcs whıch represents the ıncremental resıstance wıth respect to the length of the arc Consıder a node whıch ıs conne ted through a number of such arcs Movıng that node towards rıght wıll reduce some arc length and ıncrease the other The ımmedıate conclusıon ıs to move that node as much towards rıght as possıble ıf such movement results ın a decrease ın the total resıstance We defıne the cost parameter for each node as follows

Definition For a node v the cost of v (C_v) is defined as the inc ease in the r sistance per movement of the node v towards right by unit distance

So C is the sum of the costs of arcs which connect the node to some other nodes i e $C = \sum(C \quad C_{u})$ for all nodes u connected to v So the cost function for optimization is now linearized and can be iven as

$$Z = \sum C \times X$$

Consider a component T_k containing nodes v_l v v If we mo this component as a whole by unit distance the net ncrease in cost will be qual to the incre ise in cost of indiv idual nodes and can be given by

$$C(T_k) = \sum_{v_l \in T_k} C_l$$

3 3 3 The Algorithm

The optimization problem is solved using the simplex algorithm sp ialized for handlin linear constraints modelled by graphs The input to the optimizat on problem is the feasible solution set (feasible spanning tree) obtai d n the pre ous stage This feasible tree serves as the starting solution for simplex algorithm In th s section ve will describe the optimization procedure in detail Following is the algorithm

```
Procedure Optimize_Layout(T G)
Be in
    1 Determine the lon est path from the source node (S) to the sink node (T)
    Mark the nodes that fall on the longest path as critical nod s and the arcs on
    the longest path (critical path) as critical arcs
    for (each node $v \in G$)
        Begin
```

```
            if(v is not a critical node)
            Begin
            Determine the cost of moving that node by unit distance towards right
            End if
        End for
        for (each arc $e_j (v_i v_j) \in T$)
        Begin
            if($e_j$ is not a critical arc)
            Begin
                1 Partition the tree T into two subtrees T1 and T1 by such that $v_i \in T1$
                and T1 contains the source node and $v_j \in T1$
                2 Compute the cost of moving T1 as a whole by unit distance which is
                equal to sum of costs of node v $v \in T1$
                3 if (cost $\leq 0$)
                Begin
                Determine the arc of G which offers minimum slack for the movement
                of T1 Add that edge to the original tree and remove edge ($v_i$ $v_j$) from
                the tree Make suitable modifications in the original graph G
                Recompute the distances of various nodes
                End if
            End if
        End for
End
```

In the above algorithm we are considering one component at a time Each component is defined by one of the non critical edge of the feasible tree If we remove a non critical edge $e_j(v_i\ v_j)$ from the tree T the tree will be partitioned into two subtrees T_1 and T_1 Assume that the node v_i belongs to the subtree T_1 which also contains the source node *(S)* and the node v_j belongs to the other subtree (T_1) Note that all the critical nodes will alw ays remain in the subt ee T_I The nodes belongin to T_I will remain undisturbed for the current pass and the nodes belon in to T_I will exploit the possibility of movement to reduce the cost function

Th cost of moving all the nodes in T_I b unit distance as a whole is consid red If this cost is nonpositive it impl es that it is beneficial to move the component The next step to be considered is the maxim l distance by which w e can move this component T_I Tl e interaction of the set of nodes in T_I vith the rest of nodes vill decide this maximal distance The component T_I mav have different slacks vith respect to the nodes in T_I ho ever t is the minimum slack that n eds to b considered This is becai se ve cannot move a component by a distance more than th minimum ivailable slack The correspondin edge must b used to merge the two subtre s T_I and T_I into a single tree and the edge $e_j(v\ v_j)$ is to be removed If the cost of moving the component as a whole is positive the ed e $e_j(\ v_j)$ is not emoved from the initial tree but marked as visited and ill not be considered further When all the non critical edges are visited the algorithm terminates and the resultant tree obta ned corresponds to the optimal solution

An example of optimization process is sho vn in Fi ures 4 35 and 6 Figure 4 d scribes the initial constraint raph in hich the ed e veights and edge costs are sho vn The cost function is gi en by $Z = 10\ (X_6\ \ X_3) + 10(X_4\ \ X_1) + 100(X_7\ \ X_4)$

Figure 35 represents one basic feasible solution to the problem The positions of various nodes as per this solution are sho vn The cost of this solution is calculated to be ??50 The longest path from source to sink is sho n in dashed line The nodes $v_0, v\ \ v_5\ v_7$ and v_8 are in the longest path hence are the critical nodes The positions of these nodes are invariant to the optimization process The nodes $v_1\ v_3\ v_4$ and v_6 have some freedom for movement towards right The cost of moving these nodes are $C_1 = 10\ C_3 = 10\ C_4 = 90$ and $C_6 = 10$ We can immediatel conclude that by moving nodes $v_1\ v_3$ and v_4 towards right will reduce the cost Moving v_6 will increase the cost, hence it will not be moved

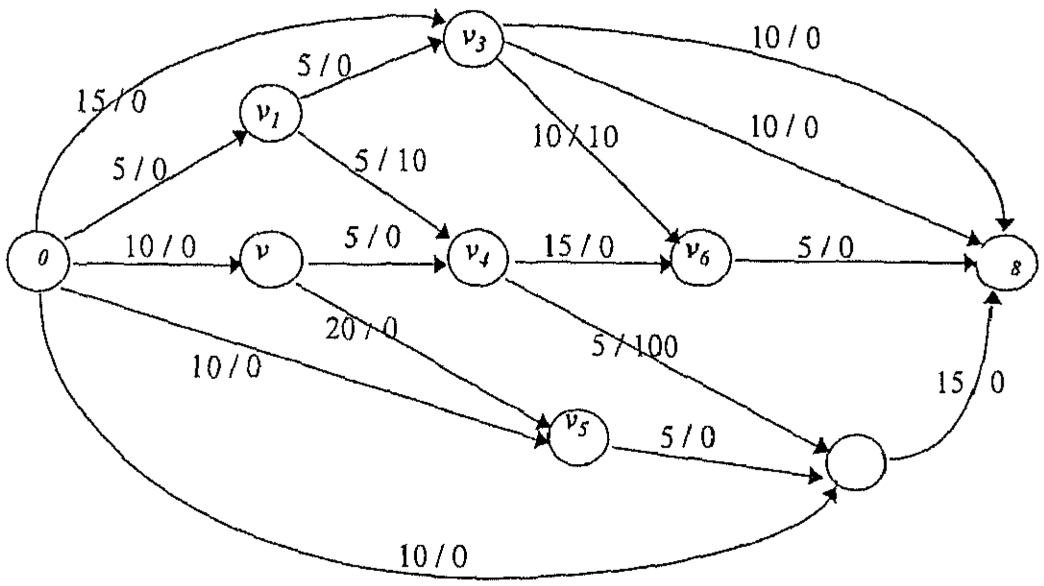


Figure 34 An Example of constraint raph Edge weights and edge costs are sho vn

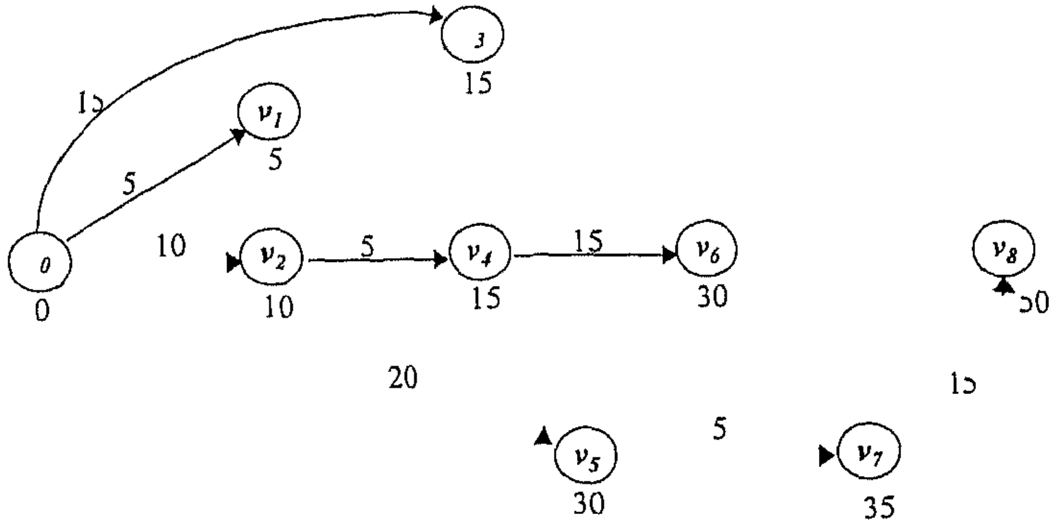


Figure 35 The *basic feasible solution* to the problem The locations of all the nodes are shown The cost corresponding this solution = 2250

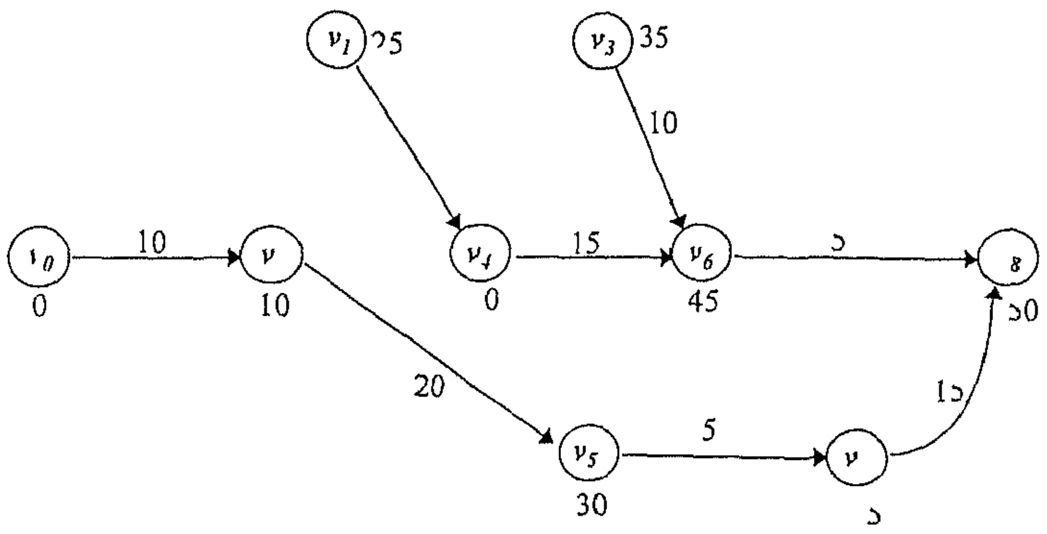


Figure 36 ***Optimal solution*** to the problem The positions of various nodes are shown Cost of this solution = 650

Although mov ng the node v_4 reduces the cost its movement s restri ted by the node v_6 But when the edge $(v\ v_4)$ is considered the nodes $_4$ and $_6$ rema n n sam component and the effecti e cost of moving the component as a whole = 80 and there is some slack for the movement The edge $(\ _6\ v_8)$ provides the minimum slack Hence the edge (v_6, v_8) is added and $(v_2\ v_4)$ is removed Figure 6 shows the tree corresponding to the optimal solution The positions of various nodes are sho n The cost of optimal solution = 650 which shows significant reduction in the cost compared to the solution obtained by solving the longest path problem

Chapter 4

Results Conclusions and Scope for Future Work

4 1 Results

In order to validate the applicability of the proposed algorithms all th mentioned algorithms are impl mented using *C* programmin lan uage and run on *UNIX* based ***HP 9000*** workstation A number of experiments ha e been made on some of the existing cells of ***ARCUS T chnology Limited*** with a set of design rules specified in [3] All the layouts are retargetted to a 0 5 micron process The arious mask layers and the detailed listing of all the design rule applicable for this process are given in [3] We took the sheet resistance (ohms/square) as the characteristic parameters pertinent to a mask layer which ar provided to us by ***M/s ARCUS T chnology Ltd*** Table 4 shows values of these parameters Table 4 1 sho vs the results obtained for different t st cases and Table 4 2 depicts the time taken by various steps for those test cases From these results it is evident that the constraint generation phase takes the maximum time and is as expected This is because of the extra computation necessary to generate as few irredundant constraints as possible which in turn speeds up the latter stages of layout recycling

Table 4 1 Performance analysis of TECHMIG for various test cases

Parameters of comparison	Layout #1	Layout #2	layout #3
No of primitives	9	30	39
No of nodes in HCG	3	70	126
No of nodes in VCG	23	70	126
No of arcs in HCG	38	253	488
No of arcs in VCG	40	257	565
Input cell size (μ^2)	11 x 6	28 x 26	13 8 x 2 8
Output cell size	8 3 x 4 4	12 9 x 18 6	12 1 x 19 6
Cost function value before optimization	13698	2395283	4366063
Cost function value after optimization	5175	2394110	3760058

Table 4 Timin analysis of the various stages of *TECHMIG* with respect to arious test cases shown in Table 4 1

CPU time in second					
Input Layout	I/O Processing	C G Generation	Obtaining a b f s	Wire Length Optimization	Total Time
Cell # 1	0 17	0 05	0 04	0 01	0 27
Cell # 2	0 24	0 98	0 02	0 03	1 27
Cell #	0 31	8 01	0 36	0 13	8 81

Tnble 4 3 Sheet resistance of various mask layers

Name of the layer	Symbol of the layer	Sheet resistance (Ω / □)
N diffusion	LCND	3 0
P diffusion	LCPD	2 5
N well	LCNW	740
P vell	LCPW	740
Polysillicon	LCP	2 8
Contact	LCC	3 8
Metnl1	LCM	0 053
Metal2	LCM2	0 053
Via	LCC2	1 7

4 2 Conclusions

A core tool ***TECHMIG*** for basic t chnolo y migration has been developed by using newer and extending e isting algorithms for its various building blo ks The constraint graph compaction method has been adopted for design ng the important compactor block in ***TECHMIG*** In this method the topological description of a given instance of cell layout is first translated into a pair of constraint graphs called the ***horizontal constraint raph*** and the ***vertical constraint g aph*** respectively These graphs serve as a model for formulating the compaction problem as a l near program Several algorithms based on computational geometry have been employed for the topological analysis of a given layout instance and for generating the two constraint graphs corresponding to irredundant constraints These have been discussed in details A graph theoretic version of the ***simplex method*** has been coded to solve the optimization problem with the minimum delay wire length as its objective function

This objective function is modelled as the weighted sum of the wire lengths *TECHMIG* has been tested on a few cells from an existing cell of *M/s ARCUS Technology Limited* The retargetted cells are obtained without any user intervention In each case the final cell is not only smaller it also has the total wire length optimized for minimum wire delays

4 3 Scope for Future Work

We have developed a basic compactor tool *TECHMIG* Further enhancements need to be carried out in *TECHMIG* to provide it with more features and flexibility so that it can result in more compacted layouts which have better timing performances The present work can be extended to include a number of issues as given below

1 Hierarchy Preservation Large industrial designs generally include hierarchical descriptions of cells To aid the design of larger cells the present work can be extended to support hierarchical descriptions of the input cells The technology migration of such cells can again be done in an hierarchical manner

2 Automatic Jog Insertion/Deletion In a practical layout there may exist few areas where a straight wire can be jogged to reduce the overall area of the layout Also there may exist some zigzag portions of a wire which can be straightened to improve the timing performances of the cell by reducing the net wire length without affecting the overall layout area Such considerations have not been included in the present work

3 Critical Path Breaking In optimizing a layout *TECHMIG* maintains the critical path as an invariant in the optimization phase The idea behind such a critical path indentation is that the portions of the layout which corresponds to the critical path are exactly minimum according to the current set of design rules and user constraints In general a cell can be made smaller by mov ng th objects off the cntical paths into non cntical areas thereby creatıng parallel cntical paths This feature needs to be ıncluded ın *TECHMIG*

4 Contact Optımızatıon If certaın portıons of the target layout permıts more number of contacts than ın the source lavout without ıncreasıng the overall area of the cell ıncorporatıng such extra contacts can result ın lower parasıtıc and ımproved current carrvıng capabılıtıes Such a fa ılıty can be added to the exıstıng tool for obtaın ng a better lavout A roup of parallel contacts can be merged to ether to form a super contact of a larger sıze In the final lavout thıs super contact can be slıced ınto an approprıate number of contacts of the desıred sıze

5 Plowın Plo vıng can be used ınteractıvelv to rearrange the geometrv of a cell b compactıng a sparse lavout or creatıng space ın a dense lavout to esult ın a bette fabrıcatıon yıeld The user places a *plo v* (a horızontal or vertıcal lıne segment) ınto the lavout and gıves a dır ctıon and dıstance the plow ıs to move Acco dıng to the specıficatıons the plow ıs then moved through the layout and the materıal b hınd ıt ıs stretched (ıf necessary) and the materıal ın front of ıt ıs compressed

6 M tal Layer Translatıo Present tre ds ın the layout desıgn phase of *VLSI* chıps ınvolves the use of larger number of metal layers for routıng to reduce the o erall cell sıze Thus ıt may become essentıal to mıgrate exıst ng cell layouts to a new process wıth larger number of metal layers Unfortunately none of the commercıally avaılable tools for layout conversıon or technology mıgratıon ıncorporates thıs essentıal feature

Bibliography

[1] C Mead L Conway *Introduction to VLSI Systems* Readin MA Addison Wesley 1980

[?] Neil H E Weste Kamran Eshraghian *Principl s of CMOS VLSI d s gn* Addison W sley publishing company 1994

[ ] Design rule documentation Arcus Technology Limited

[4] J F Lee A New Frame vork of Design Rules for Compaction of VLSI Layouts *IEEE transaction on Computer Aided Design of Integrat d Circ uts and Systems* CAD Vol 7 No 11 pp November 1988

[5] Bryan T Preas Michell J Lorenztti *VLSI Physical Design Automat on*

[6] N Sherwani *Algorithms for VLSI physical design automation* Kluwer Academic Publishers 1993

[7] Thomas Lengauer *Combinatorial aspects of integrated circuit layout* John Willey & Sons 1990

[8] H Shin A L S Vintcentelli C H Sequin Zone Refining techniques for IC Layout compaction *IEEE transactions on Computer Aided Design of Integrated Circuits and Systems* Vol 9 No 2 pp 167 179 February 1990

[9] Y Liao C K Wong An Algorithm to Compact a VLSI Symblic Layout with Mixed Constraints *IEEE Transaction on Computer Aided D sign of Integrated Circuits and Systems* Vol ? No ? pp 62 69 April 1983

[10] G Kedem H Watanabe Graph Optimization Technique for IC Layout and Compaction *IEEE Transaction on Comp ter Aided Design of Integrated Circ its and Systems* Vol 7 No 4 pp 1? 19 January 1984

[11] J F Lee D T Tan VLSI Layout Compaction with Grid and Mixed Constraints *IEEE tra isaction on Computer Aided Design of Int grat d C rcuits and Syst ms* Vol 6 No 5 pp 90 910 s ptember 1987

[1?] M Schlag Y Z Liao C K Wong An algorithm for optimal D compa tion of *VLSI* circuit *Integration* Vol 1 pp 179 209 198

[1] R L Kruse B P Leun C V Tondo *Data structures and program design in C* PHI Pvt Ltd 1996

[14] A M Tanenbaum Y Lan sam M J Augenstein *Data stri ctures i sing C* PHI Pvt Ltd 1997

[15] Rob rt Sedgewick *Algorithms in C* Addison Wesley publishin company 1990

[16] C H Papadimitriou K steiglitz *Combinatorial optimi ation algorithms and complexity* PHI Pvt Ltd 1997

[17] Yuji Shigehiro Takashi Nagata Isao Shirakawa Itthichai Arungsrisan chi Hiromitsu Takahashi Automatic Layout Recycling Based on Layout Description and

Lineɑr Piogrammıng *IEEE transactıon on Computer Aıded Des gn of Int grated Cıı c ııts and Systems* Vol 1ɔ No 8 pp 959 967 August 1996

[18] J Γ I ee C K Wong A Performance A med Cell Compactor with Automatic Jogs *ICEE transaction on Co nputer Aıd d D sıgn of Integrated Cırcuıts and Systems* Vol 11 No 12 pp 1495 1507 December 1992

[19] D A Pucknell K Eshraghıan *Basıc VLSI Desıgn Systems and Cı cuıts* PHI Pvt Ltd 199

[?0] Nɑrsıngh Deo *Graph theory wıtl Applıcatıons to Engınee ıng and Comput r S ıcnc* PHI Pvt Ltd 1997

[?1] S Z Yɑo C K Cheng D Dutt S Nahar Chı Yuan Lo A cell based hırarchıcal pıtchmɑtchıng compactıon usıng mınımal L P *IEEE Tı ansactıon on Comp ıter Aıded Desıgn of Integ ated Cırcuıts and Systems* Vol 14 No 4 pp ɔ23 526 Aprıl 199ɔ

[??] C K Cheng X Deng Yuh Zen Lıao So Zen Yao Symbolıc La out compactıon under condıtıonal desıgn rules *ICEE transactıon on Computer Aıded Desıgn of Integrated Cırcuıts and Systems* Vol 2 No 4 Aprıl 1992 pp 475 485

[23] E S Kuh T Ohtsukı Recent Advances In VLSI Layout *Proceedıngs IEEE* Vol 8 No 2 pp 237 263 February 1990

[?4] G Boıs E Cerny Effıcıent generatıon of dıagonal constraınts for 2D mask compactıon *IEEE Transcatıon on Computer Aıded Desıgn of Integrated Cırcuıts and Systems* Vol 14 No 9 pp 1119 1126 September 1996

[25] Chı Yuan Lo Ravı Varadarajan An O(n^1 5log n) 1 D Compactıon Algoııthm *Pı oc 27th ACM / IEEE Des gn Aı tomat on Conference* pp 382 387 1990

[?6] J Doenhardt Thomas Lengauer Algorıthm Aspects of 1D Layout Compactıon *IEEE tı ansactıon on Computer Aıded Desıgn of Integrated Cırcuıts and Systems* Vol 6 No 5 pp 86 878 September 1987

[?7] W H Wolf R G Mathews J A Newkırk R W Dutton Algorıthm for optımızın ?D symbolıc layout compactıon *IEEE Traısactıon on Computer Aıded Desıgn of Integı ated Cıı cuıts and Systems* Vol 7 No 4 pp 451 466 Aprıl 1988

[?8] W ıner Bonath Manfred Glesner Process Independent 2D Compactıon ın a Symbolıc Desıgn Envıronment pp 433 443 VLSI *89* Elsevıer Scıence Publ shers B V (North Holand) 1989

[29] J L Bentley T A Ottmann Algorıthms for reportıng and countıng geometrıc ıntersectıons *IEEE tı ansactıons on Comp ıters* Vol C ?8 No 9 pp 643 647 September 1979

[30] J L Bentely Derrıck Wood An optımal worst case algorıthm for reportıng ıntersectıon of rectangles *IEEE transactıon on Computers* Vol C 29 No 7 pp 571 576 July 1980

[31] K Q Brown Comments on algorıthms for reportıng and countıng geometrıc ıntersectıons *IEEE transactıons on Computers* Vol C 30 No 2 pp 147 148 February 1981

[32] V K Vaıshnavı Computıng poınt enclosure *IEEE transactıons on Computers* Vol C 31 No 1 pp 22 28 January 1982

[3] P S Tzeng C H Sequin *Efficient 2 D compaction for placement* Computer Science Division EECS Department University of California Berkeley Private Communications

[4] S Bangalore *Layout procedure for standard cell design* Department of Electrical Engineering Mississipi State University Private Communications

[5] K Mehlhorn S Nahar A faster compaction algorithm with automatic jog insertion *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* Vol 9 No 2 pp 158 166 February 1990

[36] J M Rabaey *Digital Integrated Circuits A design perspective* Prentice Hall Electronics and VLSI Series New Jersey 1996

[7] *DREAM Design Rule Enforcer and Manager* Sagantec Corporation